Eötvös Loránd Tudományegyetem

Informatikai Kar

Algoritmusok és Alkalmazásaik Tanszék

Valós idejű sugárkövetés OpenGL és WebGL felületen

dr. Hajder Levente

Tudományos főmunkatárs

Varga Péter Gergely

Programtervező Informatikus BSc

Budapest, 2017

Tartalomjegyzék

1. Bevezetés

Az elmúlt évtizedekben a számítógépek az emberiség történelmének legrohamosabban fejlődő termékei lettek. Az ehhez hasonló, már-már közhelyessé vált kijelentések hallatán a legtöbb embernek a számítógépekben található központi feldolgozóegységek (angolul: *Central Processing Unit*, röviden: *CPU*) fejlődése juthat eszébe, és többnyire igazuk is van, hiszen kulcsfontosságú szerepük van a számítógépek működésében, és rengeteget fejlődtek a 60-as, 70-es évek óta. Azonban ha csak a legutóbbi pár év kínálatát nézzük, észrevehetjük, hogy sem a CPU-k órajele, sem a bennük található fizikai, illetve logikai magok száma, sem pedig az energiafogyasztásuk nem változott jelentősen. Egy laikus ember ezekből arra következtethet, hogy az elmúlt években nem is fejlődtek a CPU-k. Ez természetesen nem igaz, hiszen – bár talán ezek egy processzor legismertebb specifikációi – nem csak ezek a paraméterek jellemzik őket.

Fejlődésük az utóbbi időszakban főleg az utasításkészletük (angolul: *instruction set*) bővítésében, okosításában látható, amivel az érhető el, hogy egyetlen utasítás ciklus alatt egyre több utasítást elvégezhessenek. Hasonló (ár)kategóriájú processzorok *teljesítményvizsgáló programokkal* (angolul: *benchmark*) végzett tesztje alapján[1] a gyakorlatban ez kb. 5 év alatt 25-35%-os fejlődést mutat. A fejlődés tehát nem állt meg, azonban ezek a számok jelentősen alulmúlják a Moore-törvénytől várt eredményeket: a CPU-k paramétereinek fejlődése határozottan lassulni látszik, kezdjük elérni fizikai korlátainkat.

Van azonban egy másik alkatrész, ami csak az ezredforduló környékén kezdett részévé válni a számítógépeknek: a grafikai feldolgozó egység (angolul: *Graphics Processing Unit*, röviden: *GPU*). Ez vagy az alaplapon, vagy újabban a CPU-n belül – lényegében minden újabb processzorban – (*integrált GPU*), vagy egy külön erre a feladatra kialakított videókártyán (*dedikált GPU*, vagy *Desktop VGA*) található meg. Természetesen a dedikált GPU-k rendszerint jóval nagyobb teljesítményűek integrált társaiknál.

[1] Az eredményeket a http://userbenchmark.com-on lévő összehasonlítás alapján írtam, ahol a 2011-es (Q1) megjelenésű Intel Core i7-2700K típusú processzort vetettem össze a 2015-ös (Q3) megjelenésű Intel Core i7-6700K-val.

A GPU-k (a CPU-kkal ellentétben) napjainkban is rohamosan fejlődnek. Ez a fejlődés sokkal jobban közelíti a Moore-törvénytől elvárt szintet: ha a 9800 GT, és a GTX 1080 típust vesszük alapul (melyek a megjelenésük idejében mind a ketten a csúcskategóriát képviselték), azt mondhatjuk, hogy az elmúlt 8 év alatt kb. 25-ször gyorsabbak lettek.[36] Ráadásul egyre szélesebb körben használhatóak, egyre több feladatot képesek átvállalni a CPU-tól, és a bennük lévő többszáz, vagy akár többezer számítási egységnek köszönhetően bizonyos – jellemzően párhuzamosan futtatható – feladatokkal sokkal (jellemzően 50-szer, de akár még többször)[2] gyorsabban végeznek, mint egy velük egy kategóriában lévő, hasonló korú CPU.

A GPU-kban rengeteg lehetőség van, amikkel a képzés során a *Számítógépes grafika* tárgyban találkoztam először. Az egyik „nagybeadandó" feladatnál egy GPU-n futó programot kellett megvalósítanom, és bemutatnom. Annyira megtetszett a dolog, hogy még abban a félévben eldöntöttem, hogy majd a szakdolgozatom témája is ezzel kapcsolatos lesz.

Szakdolgozatomban egy (főként) GPU-n futó program megtervezését, és megvalósítását mutatom be. A program lényege, hogy fénysugarakat szimulál egy téglalap minden egyes képpontjára (pixelére), és matematikai egyenletek segítségével metszéspontokat keres különböző, virtuális térben definiált geometriai alakzatokkal (gömbökkel, síkokkal, háromszögekkel, stb.). Az így kapott metszéspontokat figyelembe véve jeleníti meg az említett testeket.

A kirajzolt alakzatokra ezután árnyékolást is implementálok (angolul: *shading*), hogy élethűen nézzenek ki, sőt némelyik textúrát, vagy egészen különleges anyagi tulajdonságokat (angolul: *material*) is kap, amik azt az érzést kelthetik majd a felhasználóban, hogy az objektum aranyból, ezüstből, vagy akár üvegből van. Kitérek az ezekhez szükséges fényvisszaverődési, illetve fénytörési törvényekre, szabályaikra; bemutatom a beépített, illetve általam megvalósított függvényeket.

A program futása során egér és billentyűzet segítségével mozoghatunk majd a térben, valamint lehetőség lesz néhány tulajdonság futásidőben történő

[2] A számítógépek teljesítményének méréséhez sok esetben *FLOPS*-ot használnak (***Fl****oating-point* ***O****perations* ***P****er* ***S****econd*, magyarul: lebegőpontos műveletek másodpercenként).
Intel Core i7-4770K: ~180 GFLOPS
Nvidia GTX 1080: ~8.9 TFLOPS
8.9 TFLOPS / 180 GFLOPS = ~ 49.4

megváltoztatására, mint pl.: maximum hányszor pattanjon/törjön a fénysugár (sugarak mélysége), legyenek-e vetített árnyékok, stb.

A programot először az OpenGL grafikus szabványt használva valósítom meg, majd ezt írom át (*portolom*) WebGL-re, ami tulajdonképpen az OpenGL kistestvérének tekinthető. Ezután a programom segítségével összehasonlítom a két platformot teljesítmény, használhatóság, és egyéb szempontok alapján.

Célom, hogy ezzel a programmal demonstráljam a GPU-kban rejlő óriási potenciált, hiszen egy ilyen sugárkövető program (angolul: *Ray Tracing*) valós időben való futtatására a GPU-k megjelenése előtt esély sem volt (csak külön erre a célra kifejlesztett hardverekkel), CPU-n futó változatok esetében egy-egy képkockára jellemzően hosszú perceket, vagy akár órákat kell várni.

2. Felhasználói dokumentáció

2.1. Rendszerkövetelmények

Ebben a fejezetben bemutatom a program futtatásához szükséges minimális, és ajánlott hardveres, illetve szoftveres követelményeket, a különböző platformokra bontva.

2.1.1. Az OpenGL verzióhoz

A verziószámot az alapállapothoz (3.0) képest sikerült lejjebb vinnem, így már akár csupán OpenGL 2.1-et támogató hardveren is lehetőség van futtatni a programot, tehát a GPU-nak támogatnia kell legalább az 1.20.8-as pixelshader verziót. Ez 2006 szeptemberében jelent meg, így akinek a számítógépét ez után gyártották, annak jó esélye van arra, hogy a program megfelelően fog futni (habár valószínűleg szaggatni fog, ha a gép nagyon régi, vagy gyenge). A GPU telepített *illesztőprogramja* (angolul: *driver*) legyen a lehető legfrissebb. Amennyiben Windows alatt forráskódból szeretnénk fordítani, és futtatni, azt majd a *Számítógépes Grafika* című tárgyban tanultak alapján mutatom be. Ehhez (a fentieken kívül) feltétlenül szükséges egy *Microsoft Windows* operációs rendszert futtató számítógép (a Windows 7 a legrégebbi verzió, amivel teszteltem, annál újabb verziókkal is természetesen működik), valamint a *Microsoft Visual Studio* fejlesztői környezet (2013-as, és 2015-ös verziókkal lett csak tesztelve).[3]

Ezeken kívül nincs minimális követelménye a programnak, de természetesen ajánlott minél újabb, és minél erősebb hardveren próbálkozni, ezzel elkerülve, hogy a kép jelentősen szaggasson. Egy (2013 körül gyártott) középkategóriás laptop, integrált GPU-val már bőven elég ahhoz, hogy élvezhető sebességgel fusson a program. Azonban ha nagy felbontáson, legalább 60 képkockát szeretnénk látni másodpercenként (angolul: *Frames Per Second*, röviden: *FPS*), ahhoz már elengedhetetlen egy erősebb, dedikált GPU, különösen a „pro” változathoz, mivel abban jóval több elem van.

[3] A Windows és a Visual Studio fejlesztői környezet különböző verzióinak különböző követelményei vannak. Ezeknek utána lehet nézni a http://microsoft.com, illetve a http://visualstudio.com weboldalakon.

2.1.2. A WebGL verzióhoz

Nagyon nehéz meghatározni, hogy pontosan melyik GPU-k futtatnak WebGL-t, de nagy általánosságban elmondható, hogy az újabbak (~2011 után gyártottak) mind támogatják. Nincsen megkötés operációs rendszert illetően, a WebGL-t jellemzően még akár okostelefonunkon, vagy táblagépünkön is élvezhetjük minden további teendő nélkül. A hardveres követelményeken (és a hozzájuk tartozó illesztőprogramokon) kívül elég egy modern webböngésző. Ezek a teljesség igénye nélkül a következők lehetnek:

- Google Chrome (Chromium)
- Mozilla Firefox
- Safari
- Opera

A böngészők frissebb verzióiban a WebGL tartalmak megjelenítése alapértelmezetten engedélyezve van, így nincs szükség külön bővítmények (angolul: *plugin*) letöltésére és telepítésére, sőt még a beállításokban sem kell konfigurálnunk semmit. Ha kétségünk támad, látogassunk el a http://get.webgl.org oldalra, itt azonnal tájékoztatást kapunk a támogatásról. A futási teljesítményt, élvezhetőséget illetően ugyanaz érvényes, mint az OpenGL-nél: minél nagyobb teljesítményű az eszközünk, annál élvezetesebb a program használata.

A tesztelések során a Firefox töltötte be legstabilabban a programot, a többi böngészőnél sajnos több esetben (jellemzően Windows alatt Chrome böngészővel, WebGL 1.0-át használva) a hosszabb betöltési idő miatt egy bizonyos böngészőbe épített biztonsági API megszakította a kapcsolatot a videókártyával. Ez a WebGL kontextus elvesztését, végül a program leállását eredményezte.

Szerencsére ma már a modern, népszerű böngészők jelentős részében elérhető a WebGL 2.0-ás verzió. Ez azonban e sorok írásakor még csak kísérleti stádiumban van, emiatt külön aktiválni kell. Remélhetőleg hamarosan minden böngészőben alapértelmezett lesz.

A WebGL 2.0 aktiválása:

- https://wiki.mozilla.org/Platform/GFX/WebGL2 alapján (Firefox esetén)
- chrome://flags, majd a *WebGL 2.0 Prototype* engedélyezése (Chrome, Chromium, Canary esetén)

A WebGL 2.0 előnye az 1.0-ával szemben, hogy az ezt használó kód jobban hasonlít a natív verzión futó kódra, ugyanis az 1.0 több olyan nyelvi elemet nem támogat, amit én a natív verzióból át szerettem volna emelni. Ezeket helyettesítenem kellett alternatív megoldásokkal. E mellett mind a natív, mind a WebGL 2.0-ás verziókon egy automatikus generátor által optimalizált kód fut (lásd: 3.10), amely főként a fordítási/linkelési időn javít a webes verziókon. Ez elősegíti a program jóval gyorsabb betöltését, így azzal a bizonyos biztonsági API-val sincs gond: probléma nélkül betölt Chrome böngészőn is. Így ha van lehetőségünk, mindenképpen ajánlom a WebGL 2.0 bekapcsolását.

A webes verzió vázát Kamaron Peterson oktatóvideói[33] alapján készítettem, felhasználva Brandon Jones JavaScript alapú mátrix-, és vektorkönyvtárát (*glMatrix*).[34]

2.2. Telepítési (üzembehelyezési) útmutató

Ebben a fejezetben bemutatom a program rendeltetésszerű használatához szükséges lépéseket a különböző platformokra.

2.2.1. Az OpenGL verzióhoz

2.2.1.1. Telepítés *installer*rel (Windows)

Indítsuk el az általunk választott tetszőleges telepítőt („sima” (lite), vagy „pro” változat), majd kövessük a megjelenő utasításokat. Ha sikeresen végeztünk a telepítéssel, az asztalon létrehozott parancsikonnal indíthatjuk a programot.

2.2.1.2. Fordítás és futtatás forráskódból (Linux)

Debian alapú operációs rendszer esetén futtassuk a főkönyvtárban lévő *install.sh* nevű *bash-script*et. Ehhez írjuk be a következőt a parancssorba:

```
bash install.sh
```

Ez feltelepít, és konfigurál minden szükséges szoftvert, és függőséget (angolul: *dependency*), amely a program működéséhez szükséges, valamint lefordítja a forráskódot, és linkeli az így létrejött ún. *object file*-okat. A programot a következő paranccsal indíthatjuk:

```
./raytracing
```

Tesztelve frissen telepített Ubuntu (16.04 LTS), Debian (8.6) és Xubuntu (16.04 LTS) verziókon. Amennyiben nem-Debian-alapú disztribúción szeretnénk futtatni, kövessük manuálisan az *install.sh*-ban lévő parancsokat (az operációs rendszerünknek megfelelő parancsokkal helyettesítve őket).

2.2.1.3. Fordítás és futtatás forráskódból (Windows)

Először is telepítsük a Visual Studio fejlesztői környezetet (angolul: *Integrated Development Environment*, röviden: *IDE*). Ennek létezik ingyenes verziója is, *Visual Studio Community 2015* néven. A telepítés után helyezzük el a projekt főkönyvtárában található *OGLPack_VS2015.zip* (kicsomagolt) tartalmát a C:\ meghajtóra. Ha nem szeretnénk a gyökérkönyvtárba tenni (vagy nincs jogosultságunk rá), tegyük nyugodtan egy almappába. A lényeg, hogy az elérhetősége (az *OGLPack* könyvtárat tartalmazó mappáé) így nézzen ki:

`C:\<útvonal>`

például: `C:\Users\JohnSmith\Desktop`

Ezután indítsuk el a Windows parancssor (angolul: *Command Prompt*, röviden: *cmd*) programot. Ehhez a startmenüben keressünk rá arra, hogy *cmd*, vagy klaviatúránkon üssük le a *Windows-gomb+R* billentyűkombinációt, ezzel elindítva a *Windows Futtatás* (angolul: *Windows Run*) alprogramot, és ide írjuk a *cmd*-t, majd kattintsunk az *OK* gombra. A frissen megjelent terminálablakba pedig gépeljük be a következő parancsot:

`subst t: c:\<útvonal>`

például: `subst t: c:\Users\JohnSmith\Desktop`

Ezzel létrehozunk egy virtuális meghajtót *T:* néven. Észrevehetjük, hogy a T:\ meghajtóban található fájlok és mappák megegyeznek a parancsban megadott útvonalon található mappaszerkezettel. Innentől kezdve a T:\ meghajtó gyakorlatilag egy hivatkozás a megadott útvonalhoz. Miért van erre szükség?

A *Számítógépes Grafika* tárgyban a Visual Studio-ban használt projektek úgy voltak beállítva, hogy a T:\ meghajtóban keressenek olyan különböző függőségeket, mint amilyen az OGLPack. Az OpenGL-es programom is egy, a tárgyon belül használatos sablonprogramból lett kifejlesztve, így a függőségek kezelése is az ott bemutatott módon történik.

Ha mindezzel megvagyunk, indítsuk el a projektet, mint forrásanyagot. Ehhez kattintsunk kétszer a projekt könyvtárában lévő *RealtimeRayTracing.sln* nevű fájlra, ez megnyitja a Visual Studio-t és benne a program forrásait. A programot az ablak felső sávjában található kis zöld nyilacskára kattintva (mellette *Local Windows Debugger* felirattal) fordíthatjuk, és indíthatjuk.

Ha befejeztük a program használatát, nyugodtan megszüntethetjük az előbb létrehozott virtuális T:\ meghajtót (bár az az operációs újraindítása során is megszűnik), ehhez a következő parancsot hajtsuk végre a cmd-ben:

`subst t: /d`

2.2.2. A WebGL verzióhoz

2.2.2.1. Futtatás internetről:

Ha nem akarjuk a forrásokat magunknak beüzemelni, csupán ki akarjuk próbálni a programot, akkor látogassunk el a http://vargapeter.info/webgl/raytracing, vagy (pro változat esetén) a http://vargapeter.info/webgl/raytracing_pro weboldalra. Ezeket bárki meglátogathatja, akinek internet elérése van. Ez az összes közül a legegyszerűbb módszer: nem igényel sem telepítést, sem konfigurációt.

2.2.2.2. Futtatás forráskódból:

A WebGL-es verzió nem használ szerveroldali elemeket, kizárólag lokálisan fut. *JavaScript* kód segítségével töltődnek be a *shaderek*, és textúrák a GPU-ra, mivel azok külön állományokban vannak (angolul: *Cross-Site Scripting*, röviden: *XSS*). Az XSS viszont biztonsági okokból nem működik lokálisan (alapesetben). Ahhoz, hogy futtathassunk egy ilyen kódot, alapvetően három lehetőségünk van:

- Ha rendelkezünk saját ún. *File Transfer Protocol* (röviden: *FTP*) webes tárhellyel, akkor egy FTP-kliens segítségével feltölthetjük oda a projekt forrásait, majd egy böngészővel (a címünket és a projekt elérési útvonalát beütve) indul a program.
- Bizonyos böngészőkben van lehetőség különböző biztonsági beállítások kikapcsolására, amivel elméletileg lehetőség van az XSS kód lokális futtatására. Ennek módja azonban böngésző-, és operációsrendszerfüggő, ami rengeteg féle lehet. Ezeknek az ismertetésétől ezért most eltekintek.
- Futtathatunk ún. lokális webszervert.
    - Windows operációs rendszer esetén ajánlom a *WampServer* nevű programot, melyet letölthetünk a http://www.wampserver.com/en/ oldalról. Ezt telepítsük egy tetszőleges könyvtárba (lehetőleg ne olyanba, ahol adminisztrátor szintű jogok kellenek). Az ebben található *www* nevű mappába másoljuk be a projekt forrásait. Bizonyosodjunk meg róla, hogy a wampserver megfelelően fut: ezt jobbra alul, a tálcán egy kis zöld hátterű ’W’ feliratú ikon jelzi.[4]

[4] Ha az ikon háttere piros, vagy narancssárga, az valamilyen rendellenességet jelez. Gyakran az okoz hibát, hogy a *Skype* ugyanazt a portszámot használja, mint amit a webszerver szeretne. Megoldás: Ideiglenesen zárjuk be a Skype-ot.

- Linux operációs rendszer használatánál az *Apache* webszervert ajánlom. *Debian* alapú disztribúció (*Debian, Linux Mint, Ubuntu, Xubuntu, Lubuntu*, stb.) esetén írjuk be terminálba a következő parancsot:

  `sudo apt install apache2`

  Red Hat alapú disztribúció (*Red Hat, Cent OS*, stb.) esetén pedig a következőt:

  `sudo yum install httpd`

  Ezután kövessük a kiírt utasításokat.

  Ha készen vagyunk, írjuk a böngészőbe a következőt: `localhost`.

  Ha a webszerver megfelelően fut, egy ehhez hasonló képet kell látnunk:

  A projektet másoljuk a webszerver könyvtárába, amely Linux esetén:

  `/var/www/html`

- Mac OS X esetén az Apache alapból fel van telepítve, nekünk csak el kell indítani. Ehhez írjuk a terminálba a következő parancsot:

  `sudo apachectl start`

  Másoljuk a forrásokat webszerver könyvtárába, amely Mac esetén:

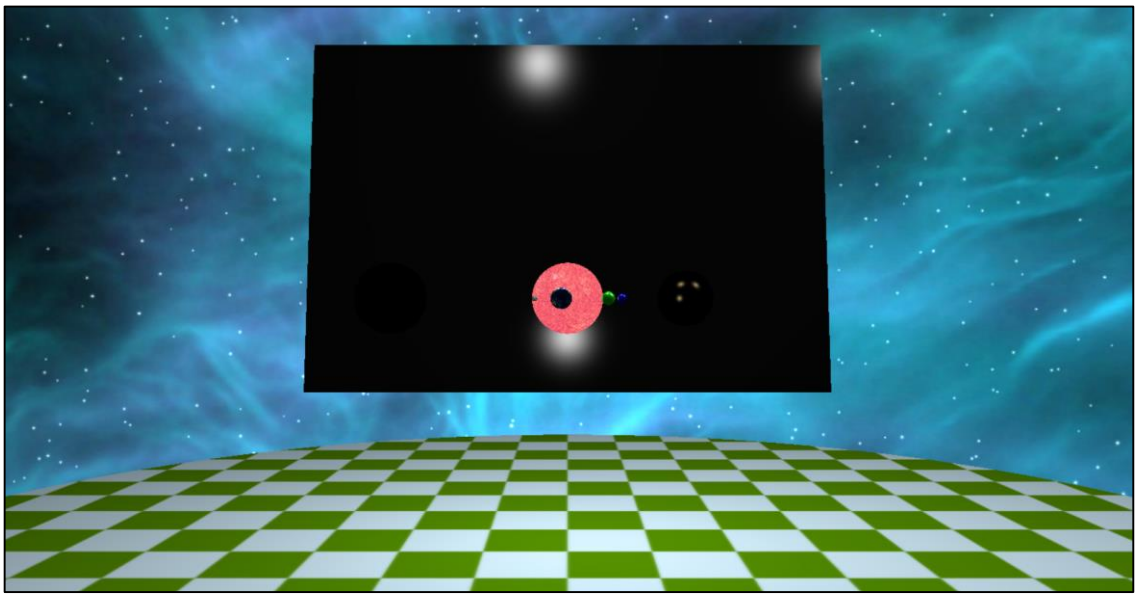
  `/Library/WebServer/Documents/`

Ha bemásoltuk a projektet, akkor a program elérhető a következő címen (melyet a böngésző címsávjába írunk):

`localhost/raytracing_webgl`

2.3. A program használata

A program indításakor a következő kép tárul elénk:



1. ábra. *Kezdőkép*

A nézet és az irányítás ún. belső nézetű lövöldözős (angolul: *First-Person Shooter*, röviden: *FPS*)[5] játékok irányítása alapján készült. A navigálás tehát a billentyűzet és az egér (mint bemeneti perifériák) segítségével, a következő módon történik:

- W: Haladás előre
- A: Haladás balra
- S: Haladás hátra
- D: Haladás jobbra
- Bal Shift: Amíg nyomva tartjuk, csökkenti a sebességünket.

Körbe nézni az egérrel a szokásos módon tudunk, annyi különbséggel, hogy az egér mozgatása közben a bal egérgombot nyomva kell tartani az alkalmazás ablakán.

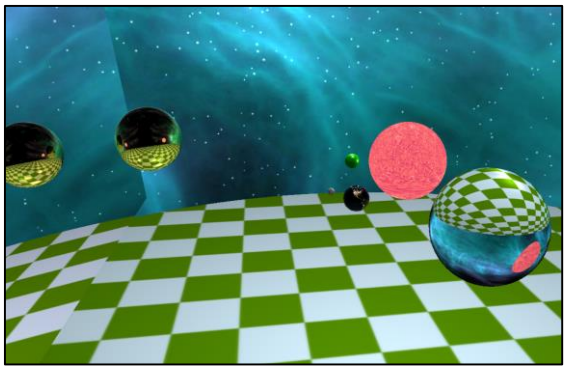
[5] Nem keverendő össze az előző fejezetben említett *Frames Per Second*-dal.

Speciális funkciókat az alábbi gombok lenyomásával tudunk állítani:

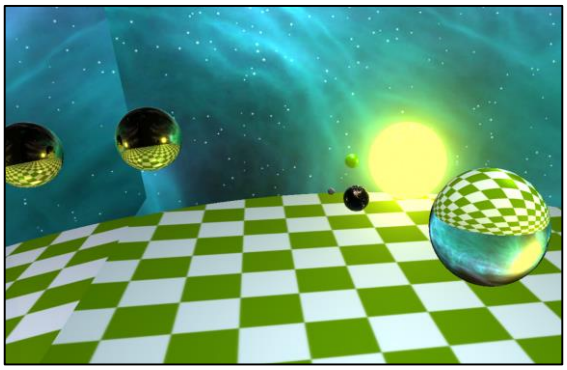
- 1: Vetített árnyékok ki-, és bekapcsolása
- G: A „Nap” körüli ragyogás-hatás ki-, és bekapcsolása (**G**low effect)
- N: *Buckatérképek*[6] ki-, és bekapcsolása (**N**ormal mapping)
- P: Az idő megállítása: az eredetileg mozgó objektumok nem mozognak. Ha még egyszer megnyomjuk, az idő újra elindul. (**P**ause)
- V: *Vertikális szinkronizáció*[6] ki-, és bekapcsolása (**V**-sync)
- →: Fénysugarak mélységének növelése (maximális: 8)
- ←: Fénysugarak mélységének csökkentése (minimális: 1)
- ↑↓: A megjelenés színmódjának (*colormode*) változtatása

A speciális funkciók használatáról a színtér ablakával együtt megnyíló terminálablakon kapunk visszajelzést. Az aktuális FPS számot az ablak fejlécében láthatjuk. A WebGL verzióknál ezeket a színteret tartalmazó html oldalon követhetjük nyomon.

A továbbiakban bemutatok néhány képet a speciális funkciók megértése végett:

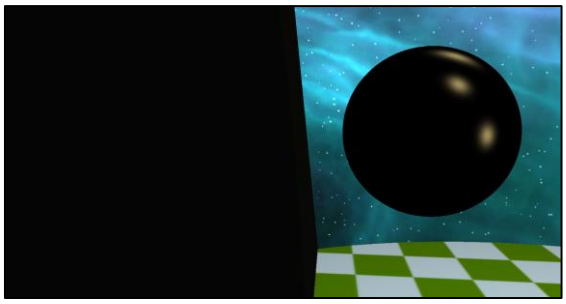


2. ábra. *Ragyogás-effekt kikapcsolva*

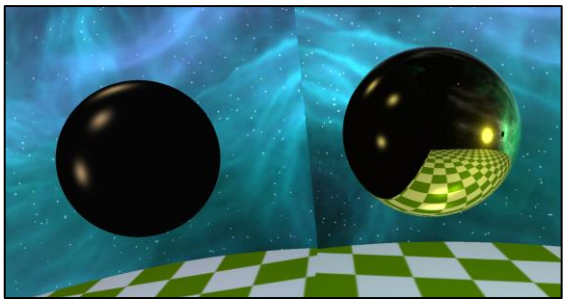


3. ábra. *Ragyogás-effekt bekapcsolva*

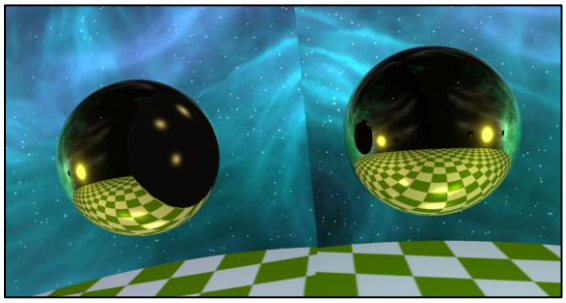
[6] Ezeknek a részletes leírására a fejlesztői dokumentációban térek ki.



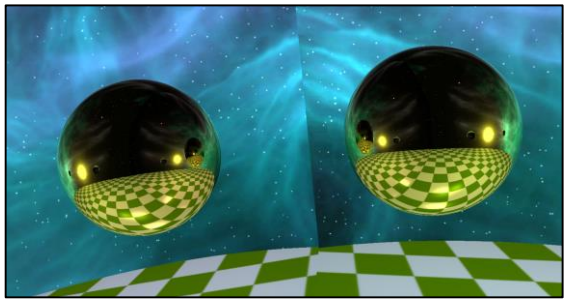
4. ábra. *Fénysugarak (max) mélysége: 1*



5. ábra. *Fénysugarak (max) mélysége: 2*

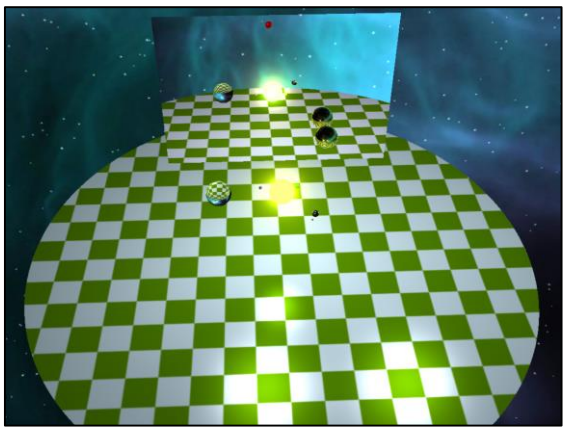


6. ábra. *Fénysugarak (max) mélysége: 3*

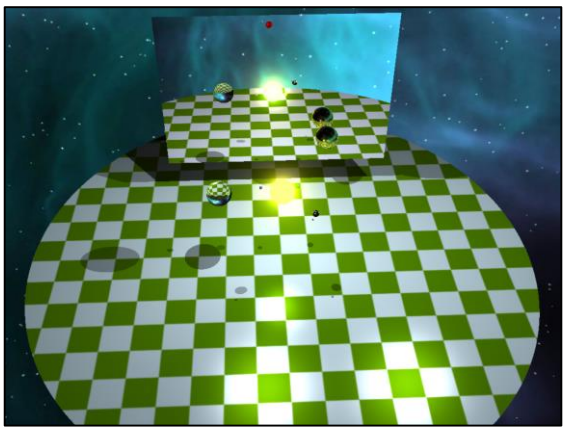


7. ábra. Fénysugarak (max) mélysége: 8

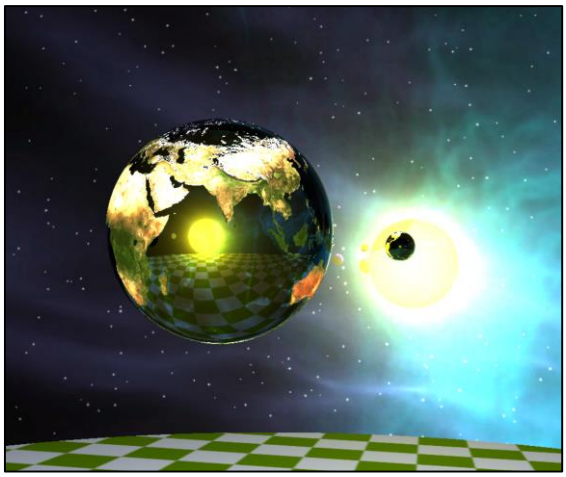
A fénysugarak mélységével azt állítjuk be, hogy a fénysugarak legfeljebb hányszor pattanjanak/törjenek. Más szavakkal fogalmazva ez azt jelenti, hogy a tükör csak akkor fog tükörként viselkedni, ha a mélység legalább 2. A tükörképben lévő tükör pedig csak akkor, ha legalább 3, és így tovább. Ez a szám 1-ről indul (a szemből „kilőtt” fénysugár az első „pattanás”), és jelenleg úgy van beállítva, hogy 8 a felső korlát, hiszen a fölött már igen nehéz észrevenni a különbséget.



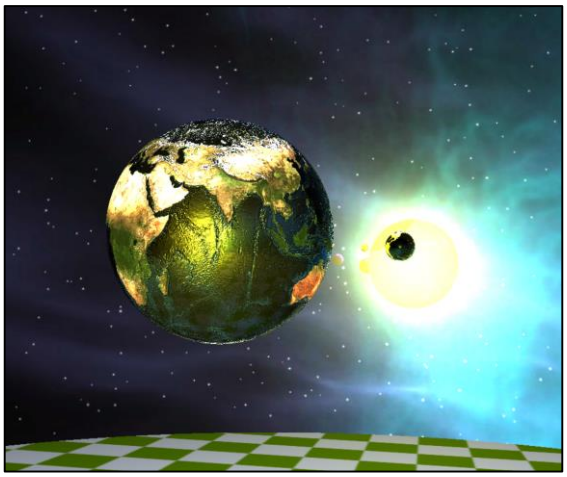
8. ábra. *Vetített árnyékok kikapcsolva*



9. ábra. *Vetített árnyékok bekapcsolva*

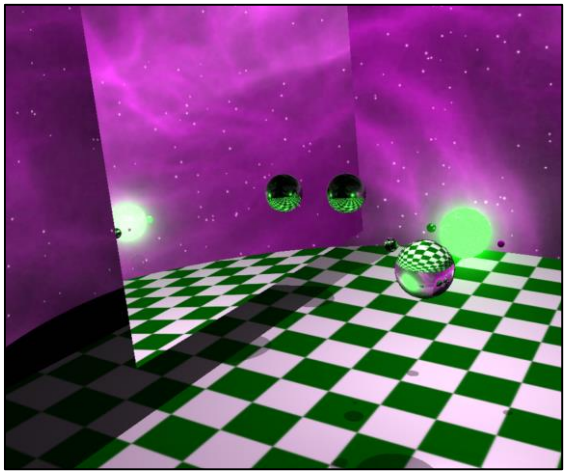


10. ábra. *Buckatérképek kikapcsolva*

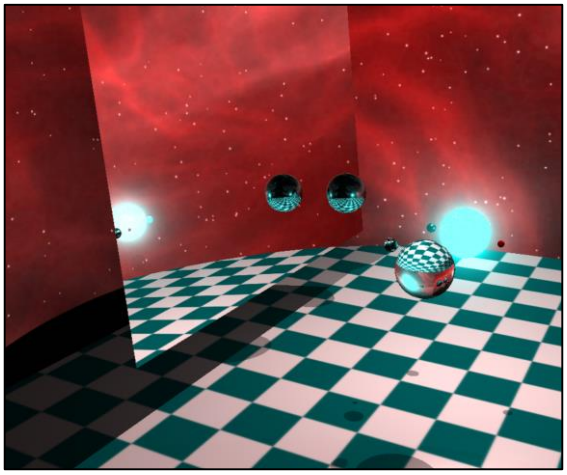


11. ábra. *Buckatérképek bekapcsolva*

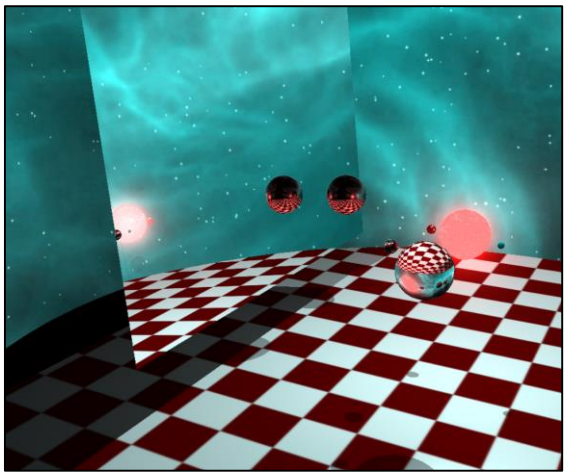
Mint láthatjuk, a buckatérképek arra valók, hogy segítségükkel göcsörtössé tegyünk egy alapvetően sima felületet. (Vagy legalábbis ezt a látszatot keltsük…)



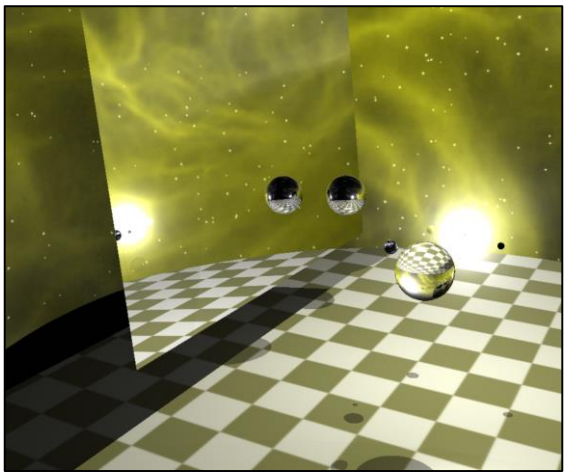
12. ábra. *BRB színmód*



13. ábra. *BRR színmód*



14. ábra. *RBB színmód*



15. ábra. *GGR színmód*

A színtér összesen tartalmaz:

- 10 db gömböt (a „pro” verziók 110-et)
- 2 db háromszöget (ezek egy téglalapot alkotnak, és tükörként funkcionálnak)
- 1 db korongot
- 6 db síkot (a csillagos-nebulás hátteret valójában 6 db sík alkotja)
- 3 db pontszerű fényforrást
- 1 db kockát (csak a „pro” verziókban van, és igazából 12 db háromszög alkotja)
- benne 1 db tóruszt (csak a „pro” verziókban)

3. Fejlesztői dokumentáció

3.1. A fejlesztéshez szükséges technológiák

Ebben a fejezetben bemutatom a programhoz felhasznált könyvtárakat, és technológiákat a kezdeti verzióktól a mai kiadásokig, valamint szót ejtek a legfrissebb, adott esetben kísérleti fázisban lévő változatokról is.

3.1.1. Az OpenGL-ről bővebben [1][2][3]

Az OpenGL (**Open G**raphics **L**ibrary) egy olyan részletesen kidolgozott szabvány, ún. alkalmazásprogramozási felület (angolul: *Application Programming Interface*, röviden: *API*), melyen keresztül lehetővé válik a grafikus processzorok (GPU-k) kezelése, valamint kettő-, és háromdimenziós vektorgrafika programozása. Kezdetben a *Silicon Graphics Inc.* fejlesztette, és adta ki 1992 januárjában. Jelenleg a *Khronos Group* non-profit konzorcium fejleszti.

Az OpenGL platformfüggetlen (erre utal az Open szó): minden mai fontosabb rendszeren felhasználhatjuk a programjainkhoz. Szinte egyeduralkodó, egyetlen riválisának talán a *DirectX* tekinthető, amely azonban csak Windowson érhető el. Felhasználása igen sokrétű: tervezőprogramok, szimulátorok, virtuális valóságok, videójátékok, sőt operációs rendszerek megjelenítése alapszik rajta.

OpenGL használatakor a programozónak viszonylag *alacsony (~gépközeli) szinten* kell kódolnia, pontosan tisztában kell lennie egy kirajzolás (angolul: *rendering*) bonyolult lépéseivel, jóformán minden alakzatot kis háromszögek sokaságából kell összerakni (tesszelláció). Ennek megvalósítása mind a programozó feladata.

Emiatt gyakran, és előszeretettel használnak sokan inkább valamilyen OpenGL-alapú, de annál jóval egyszerűbben használható *grafikus motort* (angolul: *engine*-t). Egy grafikus motor elsődleges előnye a letisztult, egyszerű használhatóság: a programozónak elég vázolnia a megjeleníteni kívánt objektumot (alakját, méretét, helyzetét, anyagtulajdonságait, stb.), és onnantól az engine-re bízhatja magát, nem kell törődnie a tesszellációs, valamint GPU memóriakezelési feladatokkal.

A játékfejlesztő cégek is általában valamilyen engine segítségével dolgoznak (sokszor saját engine-t írnak egy-egy játék írása előtt), hiszen így jóval gyorsabban lehet

haladni, valamint a kapott termék (mind a kód, mind a kész játék kinézete) egységesebb képet ad.

A grafikus motorok hátránya éppen az, mint amiből az előnyei származnak: kihagyja a programozót a lényegi részből, nem engedi, hogy benézzen (beleszerkesszen) a „motorháztető alá”, ahol a részletek vannak. Így, ha a programozó egy, az engine algoritmusától eltérő eljárással szeretne implementálni egy elvégzendő megjelenítési feladatot, kénytelen lesz egy másik motort, vagy végsősoron OpenGL-t használni, hiszen mégiscsak abban van a lehető legnagyobb szabadsága.

Az OpenGL jelenlegi legfrissebb stabil változata a 4.5-ös verziószámot kapta, és 2014 augusztusában jelent meg.

Mindenképpen említést érdemel még a *Vulkan*[4] névre hallgató API, amit szintén a Khronos Group fejleszt. A Vulkan célja, hogy a programozónak még több szabadságot, és hozzáférést adva még több optimalizációs lehetőséget biztosítson, ez által pedig gyorsabb futást tegyen lehetővé. Ezt annak árán éri el, hogy a programozónak még az OpenGL-es kirajzoláshoz képest is jóval több dologra kell odafigyelnie, sokkal több lépést kell manuálisan lekódolnia, ám ha ezeket ésszel csinálja, akkor végeredményül hatékonyabb kódot kaphat. A Vulkan a szakdolgozat írásakor még nagyon új (nincs egy éves az első kiadása), szinte gyerekcipőben jár, ám a korai tesztek alapján[7] a későbbiekben mindenképpen érdemes lesz odafigyelni rá.

Az OpenGL csak magát a rajzolást végzi el, az ablak létrehozásával, az egér, és a billentyűzet kezelésével nem foglalkozik. Ezekre a cross-platform *SDL2*, és *SDL2-image* nevű könyvtárakat használom, melyek egy köztes réteget képeznek az OpenGL context, és az operációs rendszer között: velük együttműködve hajtják végre az esemény-, és az ablakkezelést. (GLUT-ról szóló írás alapján [5])

[7] Az említett tesztben a legendás *DOOM* videójátéknak a régóta várt 2016-os verzióját vették górcső alá. A játék ugyanis az elsők között támogatta, hogy a felhasználó váltogathasson az OpenGL, és a Vulkan használata között. A teszt eredményei elérhetőek a következő címen: http://www.pcgamer.com/doom-benchmarks-return-vulkan-vs-opengl/2/

3.1.2. A WebGL-ről bővebben [6][7]

A WebGL (**Web G**raphics **L**ibrary) grafikus programkönyvtár szintén a Khronos Group kezelése alatt van. Kifejezetten a JavaScript programozási nyelvet hivatott kiegészíteni 3D-s grafikai lehetőségekkel. Mint azt már az előző fejezetben említettem, megfelelő böngészőt használva ez jóformán egyedülálló módon lehetőséget biztosít – bármiféle bővítmény nélkül is – háromdimenziós grafika megjelenítésére egy weboldalon. Voltak már a WebGL előtt (sőt jelenleg is vannak) egyéb próbálkozások webes 3D tartalmak megjelenítésére, azonban feltehetően a bonyolultságuk, zártságuk, valamint a támogatottságuk hiánya miatt a WebGL gyakorlatilag valódi egyeduralkodóvá nőtte ki magát. Az első verziója 2011 márciusában jelent meg.

A WebGL alapjául az *OpenGL ES* (***OpenGL** for **E**mbedded **S**ystems*) szolgál, amely lényegében részhalmaza az eredeti OpenGL-nek, de ezt főleg olyan, ún. *beágyazott rendszerekhez* (angolul: *embedded systems*) használják, mint az okostelefon, a táblagép, a játékkonzol, vagy akár az *egykártyás számítógépek* (angolul: *Single-Board Computer*, röviden: *SBC*). A WebGL használata rendkívül hasonló az OpenGL-éhez: egy OpenGL-ben írt kisebb méretű program általában viszonylag gyorsan, és egyszerűen átmásolható WebGL-re, csupán a szintaktikai eltérésekre kell odafigyelni, amik főleg a JavaScript, és a C/C++ nyelvek különbségeiből fakadnak. Az ablak-, és eseménykezelést a webes verzióban a használt böngésző, és vele együttműködve a JavaScript nyelv végzi.

Itt fontosnak tartom megemlíteni, hogy várhatóan pár éven belül még ennél is egyszerűbb lesz PC-re írt 3D-s programot weben futtatni, ugyanis fejlesztés alatt áll egy *WebAssembly* nevű projekt[8], melynek elsődleges célja a C, és C++, majd később egyéb programozási nyelvek böngészőn belüli támogatása. Egyesek szerint a WebAssembly megjelenése drasztikus változásokat fog okozni, fontos mérföldköve lesz az informatika, ezen belül főleg a webböngészés történetének. Olyan neves cégek emberei is dolgoznak a projekten, mint a *Google*, *Apple*, *Microsoft*, vagy a *Mozilla*. E sorok írásakor a WebAssembly még csak kísérleti stádiumban van.

A grafikus motorokat illetően itt is hasonló a helyzet, mint OpenGL esetén: rengeteg *keretrendszernek* (angolul: *framework*) az alapjává vált. A WebGL 2.0-ás verziója jelenleg kísérleti fázisban van, ám a programom képes ezt is használni (lásd: 2.1.2).

3.2. Felhasznált törvények, fogalmak, szabályok

Mielőtt igazán megértjük, hogyan is működik egy sugárkövető (angolul: *Ray Tracer*) program, ismernünk kell néhány definíciót, fogalmat, szabályt.

3.2.1. A geometriai optika alaptörvényei

Mit látunk? Erre a kérdésre sok meghatározás létezik. Mi vegyük azt, mely szerint: **„Azokat az objektumokat látjuk, amelyekről fény jut a szemünkbe”**. De mi a fény? Anélkül, hogy nagyon belemennék a részletekbe, most csak támaszkodjunk olyan alapvető fénnyel kapcsolatos felfedezésekre, mint hogy a fény egyfajta elektromágneses sugárzás, és kettőstermészetű: hullámként, és részecskeként is viselkedik. Közvetve szükségünk lesz még a *Fermat-elv*re, mely szerint **a fény két pont között mindig azon az úton halad, melynek megtételéhez a legrövidebb idő szükséges**. Ebből többnyire levezethetőek a *geometriai optika* alaptörvényei, melyekből nekünk a következőkre lesz szükségünk:

1. A fénysugár (homogén, izotróp közegben) egyenes vonalban terjed.
2. Új közeghatárhoz érve a fénysugár részben elnyelődik, részben visszaverődik, részben pedig megtört irányban folytatja útját.
3. A fénysugarak megfordíthatóságának elve: ha egy fénysugár a tér egy meghatározott (A) pontjából egy bizonyos úton halad egy másik (B) pontba, akkor az onnan visszafelé indított fénysugár ugyanezen az útvonalon fog haladni (, és emiatt keresztül fog menni (A) ponton).

Ezeken kívül nélkülözhetetlen, hogy tisztában legyünk olyan fogalmakkal, mint:

- **Rasztergrafika:** Olyan digitális kép, mely apró, szabályos négyzetrácsban elhelyezkedő képpontok (*pixelek*) sokaságából áll, és a kép minden egyes megjelenített pixelét önállóan definiáljuk.[9]
- **RGB színtér:** Olyan három színből álló additív színmodell, amely a vörös (**R**ed), a zöld (**G**reen), és a kék (**B**lue) különböző mértékű keverésével határozza meg a különböző színeket. Jó, ha tudjuk, hogy a rasztergrafikában szereplő pixelek színét is ilyen módon definiáljuk.[10]

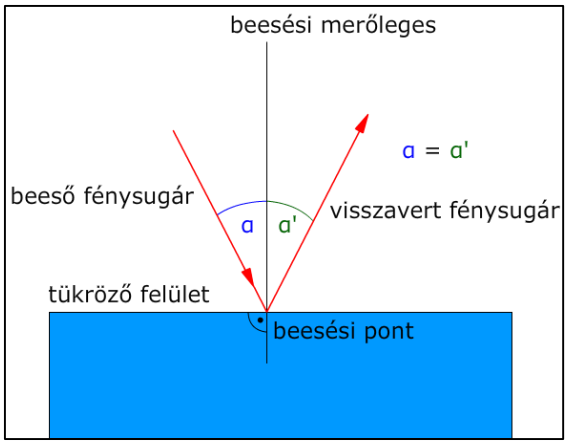
3.2.2. A fényvisszaverődés törvényei [23]

Fogalmak:

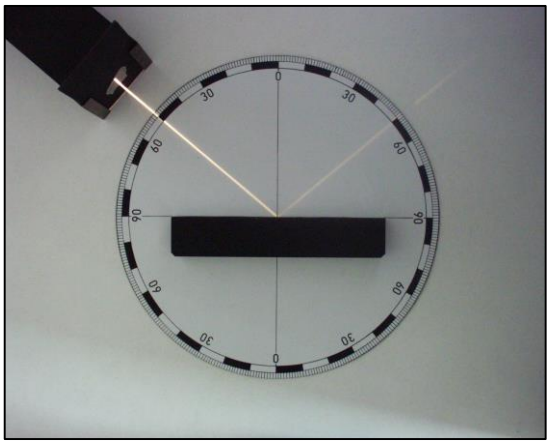
- **Beesési pont**nak nevezzük a két közeg határfelületén azt a pontot, ahova a (vizsgált) fénysugár beérkezik.
- **Beesési merőleges**nek nevezzük a beesési ponton átmenő, két közeg határfelületére merőleges egyenest.
- **Beesési szög**nek hívjuk a beeső fénysugár és a beesési merőleges közti szöget.
- **Visszaverődési szög**nek nevezzük a visszavert fénysugár és a beesési merőleges közti szöget.

A fényvisszaverődés törvényei:

- A beeső fénysugár, a beesési merőleges, és a visszavert fénysugár egy síkban van.
- A beesési szög, és a visszaverődési szög ugyanakkora.



16. ábra. *Fényvisszaverődés (illusztráció)*



17. ábra. *Fényvisszaverődés bemutatása lézerrel [24]*

3.2.3. A fénytörés törvényei [25]

Az előző fejezetben lévő fogalmakon kívül a következőre lesz szükség:

- **Törési szög**: a megtört fénysugár, és a beesési merőleges által bezárt szög.

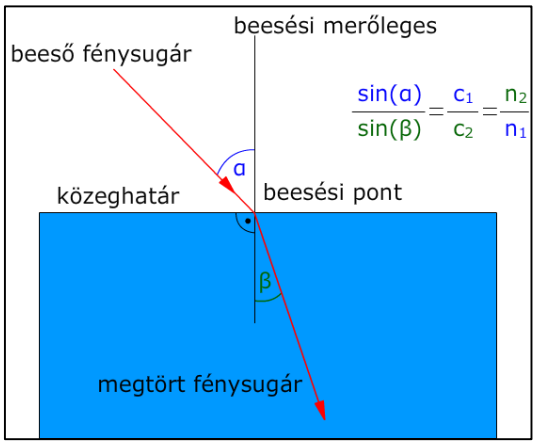
A fénytörés törvényei:

- A beeső fénysugár, a beesési merőleges, és a megtört fénysugár egy síkban van.
- A merőlegesen beeső fénysugár nem törik meg.
- A beesési szög szinuszának és a törési szög szinuszának hányadosa a két közegre jellemző állandó. (Ezt hívják *Snellius-Descartes törvény*nek is.)

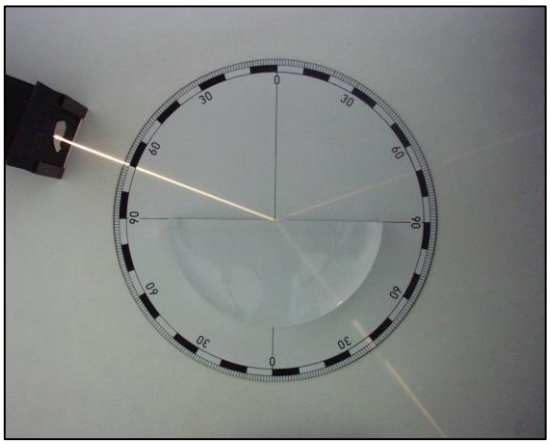
Egyéb fogalmak, tudnivalók:

- Két közeg közül az a(z optikailag) ritkább, amelyben a fény gyorsabban halad.
- Optikailag ritkább közegből sűrűbb közegbe való áthaladáskor a fénysugár a beesési merőleges felé törik.
- Optikailag sűrűbb közegből ritkább közegbe való áthaladáskor a fénysugár a beesési merőlegestől törik, amíg a megtört sugár meg nem haladja a közeghatárt (amíg a törési szög meg nem haladja a derékszöget). Amint ezt meghaladná, teljes belső visszaverődés történik (angolul: *total internal reflection*).
- A Snellius-Descartes törvényben szereplő állandót a második közegnek az első közegre vonatkozó törésmutatójának nevezzük (relatív törésmutató), jelölése n_{21}.
- Minden anyagban kisebb a fény (és minden elektromágneses hullám) terjedési sebessége, mint vákuumban. Ezt egy adott anyagra jellemző (abszolút) törésmutatóval jellemezhetjük a következőképpen: $n = c_0 / c$, ahol n a közeg abszolút törésmutatója, c_0 a vákuumbeli fénysebesség (3×10^8 m/s), c pedig a fény közegbeli terjedési sebessége.

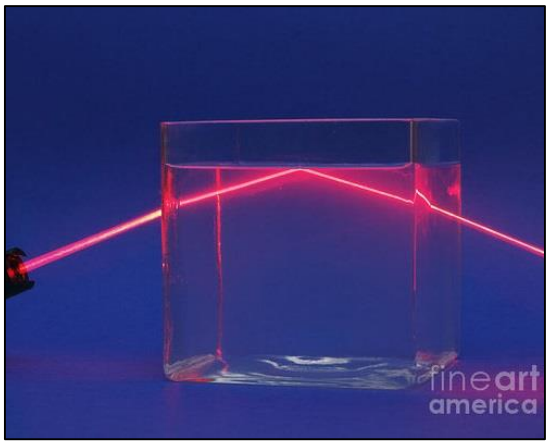
- A relatív törésmutatót (a Snellius-Descartes törvényben szereplő állandót) megkaphatjuk úgy is, ha a második közeg abszolút törésmutatójának az első közeg abszolút törésmutatójával való hányadosát, vagy a közegekben mérhető terjedési sebességek (c_1, c_2), illetve a közegekben mérhető hullámhosszok (λ_1, λ_2) hányadosát vesszük a következők szerint: $n_{21} = n_2 / n_1 = c_1 / c_2 = \lambda_1 / \lambda_2$



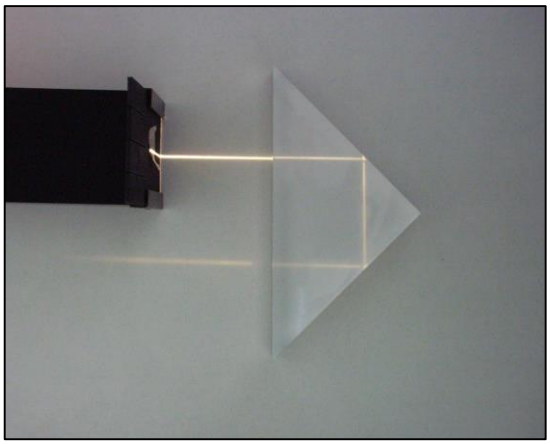
18. ábra. *Fénytörés (illusztráció)*



19. ábra. *Új közeg határán a fény részben elnyelődik, részben megtörik, részben visszaverődik [24]*



20. ábra. *Teljes belső visszaverődés vízben [26]*



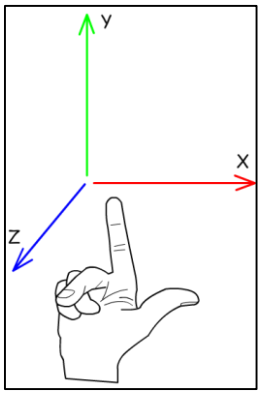
21. ábra. *Teljes belső visszaverődés derékszögű prizmában [24]*

3.3. A program szerkezete, működése

Ebben a fejezetben bemutatom a grafikus programok felépítését, komponenseinek feladatát.

3.3.1. A virtuális tér a grafikus könyvtárakban

Az OpenGL, a WebGL, sőt a legtöbb háromdimenziós platform úgynevezett *jobbkezes koordinátarendszert* használ. Ennek jelentését könnyedén megérthetjük, ha felemeljük magunk elé a kezünket az ábrán látható módon. Ekkor a hüvelykujjunk az x, mutatóujjunk az y, középső ujjunk pedig a z tengelynek felel meg. Egy térbeli pontra általában egy háromelemű vektorral hivatkozunk. A vektorokkal végezhető műveletek közül mindenképpen ismernünk kell a következőket:



22. ábra. *Jobbkezes koordinátarendszer*

- Két vektor összege. Eredménye: vektor.
- Két vektor különbsége. Eredménye: vektor.
- Két vektor skaláris szorzata. Eredménye: szám (skalár).
- Vektor szorzása számmal (skalárral). Eredménye: vektor.
- Két vektor vektoriális (másnéven *kereszt*-) szorzata. Eredménye: vektor.

A kamera látóterében folyamatosan a négyzetrács van, akármerre megyünk. Ez úgy van megoldva, hogy a négyzetrács négy sarka „rá van feszítve” egy elforgatott, téglalap alapú gúla (piramis) négy alaplapi csúcspontjára. A gúla csúcsa maga a kamera (lásd: 23. ábra).

3.3.2. A vászon (canvas) kirajzolása

Klasszikus értelemben (*inkrementálisan*) gyakorlatilag csak kettő darab háromszöget rajzolunk ki: a 23-as számmal jelölt ábrán az ABD, valamint DBC jelölésű pontok által meghatározott háromszögeket. Ezek együtt alkotják a „vásznat”, ahol a kép jelenik meg. Ennek pixeleit fogjuk egyesével kiszínezni. Egy mozgó 3D program alap algoritmusa a következő szokott lenni, ez a CPU-n fut:

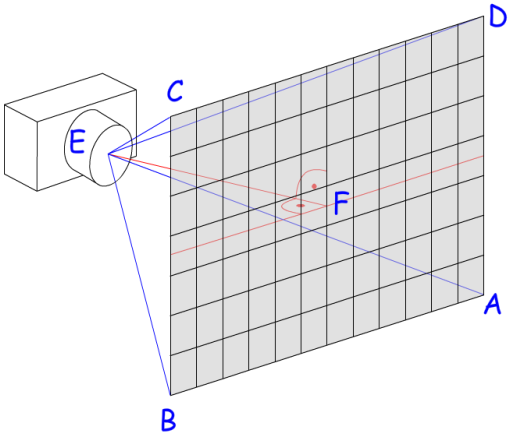
```
init();
while (!quit)
{
      update();
      render();
}
```

Ezen metódusok elsődleges szerepe a következő:

- Init(): A kamera, a kirajzolandó objektumok (háromszögek) inicializálása; textúrák, modellek betöltése, stb. Csak egyszer kell végrehajtani.
- Render(): Az előző képkocka törlése a pufferekből, a shaderprogram bekapcsolása, *uniform* változók átadása a GPU-nak, Az inicializált poligonok (többnyire háromszögek) kirajzolása, shaderprogram kikapcsolása. Ezen utasítások minden képkocka kirajzolásánál lefutnak. Egy mai átlagos (60 Hz) monitor esetén másodpercenként 60-szor (ha a hardver képes rá, lásd: 3.8).
- Update(): Minden egyéb olyan metódus, amit képkockánként le szeretnénk futtatni, ez lehet pl.: irányításkezelés, stb.

Az ABCD téglalap négy sarkát úgy definiáltam, mintha a kamera szemszögéből merőleges irányból néznénk a vásznat, ami a z = 0 síkon helyezkedik el:

- A → (-1, -1, 0)
- B → (1, -1, 0)
- C → (1, 1, 0)
- D → (-1, 1, 0)



23. ábra. *Kamera-vászon szerkezet*

3.3.3. A Vertex Shader

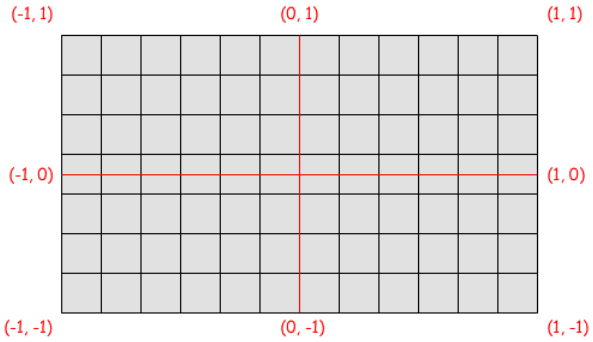
Az előző alfejezet végén definiált csúcspontokat más néven **vertex**eknek hívjuk. A vertex egy ún. *attribútum*. Mikor ezeket kirajzoljuk a *render()* függvényben, a megjelenítés előtt minden egyes vertexre egymástól függetlenül lefut az ún. **Vertex Shader**. Ez a GPU-n futó egyik olyan kód, amit szerkeszthetünk. Ennek szakszerű megírásával érjük el, hogy az előbb definiált csúcspontok mindig a megfelelő pozícióban helyezkedjenek el. A shaderek nyelve a *GLSL* (*Open**GL** **S**hading **L**anguage*). Esetünkben a Vertex Shader így néz ki:

```
in vec3 vertPosition;
out vec3 vsRay;

uniform vec3 eye;
uniform vec3 up;
uniform vec3 fw;
uniform float ratio;

void main()
{
     gl_Position = vec4(vertPosition, 1.0);
     vec3 pos = eye + fw*3.0 + ratio*right*vertPosition.x +
                      up.vertPosition.y;
     vsRay = pos – eye;
}
```

A *gl_Position* első három koordinátája a kirajzolt vertex úgynevezett NDC (***N**ormalized **D**evice **C**oordinates*) koordinátái. Ez lényegében egy 2 egység oldalú kockát jelöl (szemből nézve négyzetet), melynek oldalai a tér minden irányában -1.0 és 1.0 között vannak. Fontosnak tartom megjegyezni, hogy ezt a négyzetet (mint ablakot) elnyújthatjuk fekvő téglalappá, de ekkor a szélei ugyanígy lesznek definiálva, csak egy egység az x tengelyen hosszabb lesz, mint az y-on. Természetesen, ha álló téglalappá nyújtjuk, akkor annak megfelelően az y-on lévő egység lesz hosszabb.

Ez nekünk pont megfelel, hiszen a vászon négy sarkát ennek megfelelően határoztuk meg, tehát a vászon pont kitölti az egész ablakot. A kódban lévő *up, fw (forward), right,* valamint *ratio* változók szerepe az, hogy segítségükkel a vászon négy sarkának pozícióját (*pos*) meghatározzuk a szemhez (*eye*) képest. Ezeket a változókat a *render()* függvényben, CPU oldalon számoljuk ki, és adjuk át a GPU-nak uniform változókként, melyeknek pont az a lényegük, hogy ilyen módon a CPU, és a GPU oldal között kapcsolatot, kommunikációt teremtsenek. A fenti kód tehát lefut mind a négy

sarokpontra: például az *A* csúcs esetében a *pos*-t úgy kapjuk meg, hogy a szem pozíciójából három egységet előre, -1 egységet föl (tehát 1 egységet le), és -1 egységet jobbra (tehát 1-et balra) megyünk a képaránnyal megsúlyozva (*ratio*). A négy sarok pozíciója tehát megvan, innen már rendkívül egyszerű a kamerából az ide vezető irányt meghatározni. Láthatjuk, hogy a *vsRay* változó deklarálásánál szerepel egy *out* kulcsszó. Ez azt jelzi, hogy a mögötte álló változót (*vsRay*) a Vertex Shaderből tovább adjuk, az ún. *Fragmens Shader* fogja megkapni.



24. ábra. *NDC-koordinátarendszer*

3.3.4. A Fragmens Shader

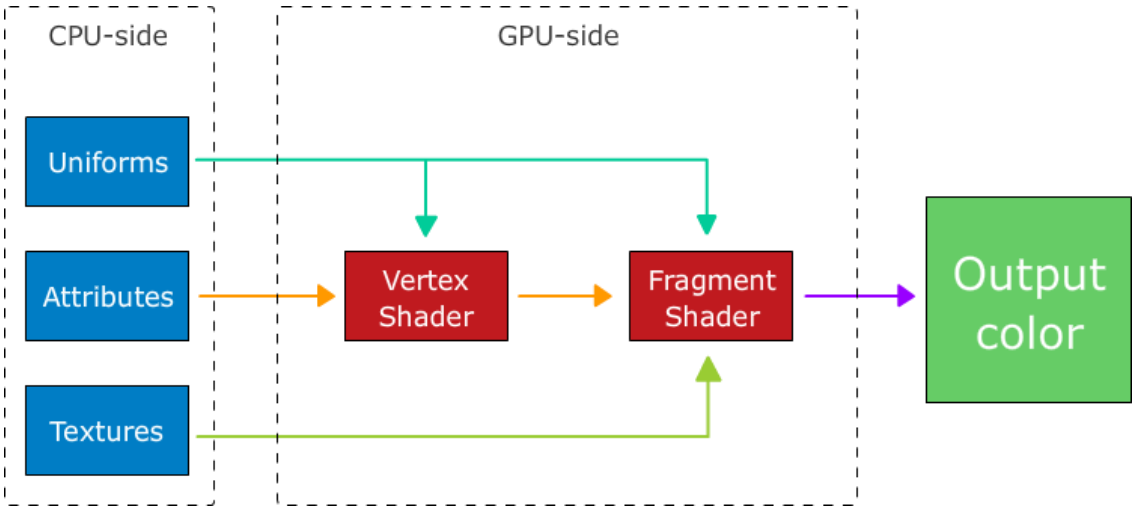
A Fragmens Shader (angolul: *Fragment Shader*, esetleg *Pixel Shader*) a másik[8] olyan GPU-n futó kód, amit programozni tudunk. Az előző ábrán láthatjuk, hogy ami a Vertex Shaderből kimenő adat, az a Fragmens Shadernél bejövő adat lesz: ezt GLSL-ben *in* kulcsszóval[9] jelöljük a bejövő változó deklarálásánál.

A Fragmens Shader az összes kirajzolt vertex minden egyes kirajzolt pixelére lefut egymástól függetlenül úgy, hogy a vertexek közötti pixelek futásakor lineáris interpolációt használ. Ennek köszönhetjük, hogy az itt már *bejövő változó*ként létező *vsRay* már nem csak a gúla négy csúcsa felé mutató irányt fogja jelenteni, hanem a négy csúcs közötti **összes** kirajzolt képpont felé mutató irányt. Ezt a grafikus könyvtár automatikusan megcsinálja, nagyban megkönnyítve a dolgunkat. A Fragmens Shadernek csupán egyetlen *kimenő változó*ja (*out*) lehet, ami maga a kirajzolt pixel színe lesz (többet

[8] OpenGL 3.2-ben megjelent az ún. *Geometry Shader*, amely a Vertex Shader és a Fragmens Shader között fut le. Ezt azonban nem használtam fel a munkámhoz, így a továbbiakban csak a két „klasszikussal” foglalkozunk.

[9] Régebbi verziókban, valamint WebGL-ben az *in*, és *out*-ra egyetlen közös kulcsszó létezik, a *varying*. Ez alól kivételek természetesen a függvénydeklarációk, melyek paramétereit bármelyik verzióban jellemezhetjük *in*-nel, *out*-tal, vagy *inout*-tal.

is definiálhatunk, de csak az első fog számítani). Ennek deklarációja opcionális, hiszen a Vertex Shaderben látott *gl_Position*-höz hasonlóan itt létezik egy beépített *gl_FragColor* nevű változó, ami szintén ezt a funkciót tölti be. Maga a sugárkövetés algoritmusa tehát itt, a Fragmens Shaderben lesz megírva, hiszen így tudjuk azt garantálni, hogy a kód a GPU-n, az ablakunk **minden egyes képkockának minden egyes pixelére** lefusson.



25. ábra. *Egyszerűsített ábra a grafikus futószalag (graphics pipeline) működéséről*

A sugárkövetéshez az elkészült Fragmens Shader kb. 800 sorból áll. Ezt minden egyes pixelre lefuttatja a GPU, és ez rengeteg számítást igényel. Szemléltetésképpen: ha *Full HD* (1920x1080 pixel) felbontáson, másodpercenként 60-szor frissül a kép, akkor az a 800 sorból álló kód összesen **2 073 600-szor** fut le **egyetlen másodperc alatt 60-szor**. Másképp fogalmazva: összesen egy másodperc alatt **99 532 800 000** (majdnem 100 milliárd!) sornyi kód fut le. Ez a gondolatsor persze nem állja meg pontosan a helyét, mivel nem egyértelmű, hogy egy sorban hány utasítás van, ráadásul a kódban ciklusok, elágazások, valamint olvashatóság miatt üres sorok is vannak. Tehát azt, hogy egy adott pixelre hány sor kód fut le, ebből csak nagyjából lehet megmondani. Mindenesetre ez nagyságrendben is óriási szám. Ezt főleg annak köszönhetjük, hogy a mai GPU-kban sokszáz, vagy akár többezer feldolgozó egység (másik nevén *shading unit*, *rendering unit*, Nvidia esetén esetleg *CUDA core*) van, és a *ray tracing* tipikusan olyan feladat, amit ezek az egységek önállóan, egymástól függetlenül, párhuzamosan képesek futtatni.

3.3.5. Mozgás a térben

Ahhoz, hogy a kameránkkal mozogni tudjunk a térben, a következő 3 elemű vektorokra mindenképpen szükség van (vagy tetszőlegesen az ellentetteikre):

- **forward**: az előre mutató vektor, alapesetben: **(0, 0, -1)**
- **right**: a jobbra mutató vektor, alapesetben: **(1, 0, 0)**
- **up**: a fölfele mutató vektor, alapesetben: **(0, 1, 0)**

Az *up* vektor (nem keverendő a 3.3.3-ban szereplő *up* vektorral) konstans. A többinek a változtatása az egér mozgatásával történik a következő módon:

```
forward = getSphereUV(u, v);
right   = normalize(cross(forward, up));
```

Ahol *u, v* változók mindig a kurzor ablakon lévő aktuális pozíciójának és az előző *frame*ben rögzített pozíciók különbsége alapján változnak (az *u* az *x* koordináta szerint, a *v* az *y* szerint), a *getSphere()* függvény pedig az ezekből kiszámolt normalizált polárkoordinátát adja vissza.

Ezt úgy kell elképzelni, hogy a kamera egy egységsugarú gömb középpontjában van, és a forward vektor mindig a gömbnek valamelyik pontjára mutat. A *v* ugyanakkor a rögzített (0.01, 3.14) intervallumon belül változhat csak, ezzel van megakadályozva, hogy a gömb tetejénél, vagy az aljánál „átforduljon”, más szavakkal ennek köszönhetjük, hogy „nem fordulhat a feje tetejére a világ”.

A WASD billentyűk működése ezek után már szinte triviális módon történik:

- W: toljuk el a kamerát forward irányba
- A: toljuk el a kamerát –right irányba
- S: toljuk el a kamerát –forward irányba
- D: toljuk el a kamerát right irányba

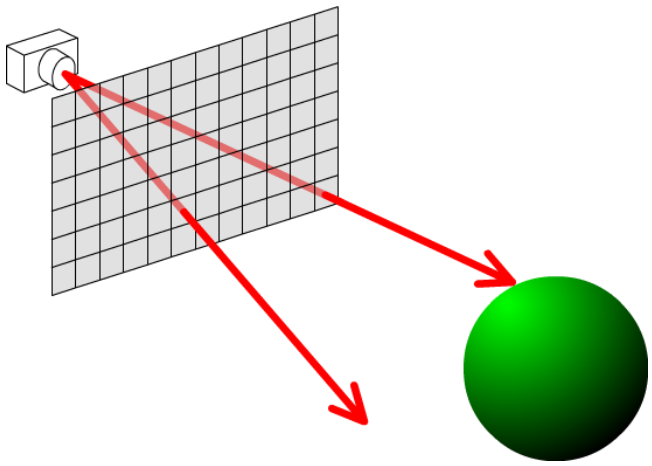
3.4. Képalkotó eljárásárok

Ebben a szakaszban bemutatom a fénysugarak szimulálásával működő képalkotó algoritmusok fejlődését a kezdetektől az én munkámig.

3.4.1. A sugárvetés

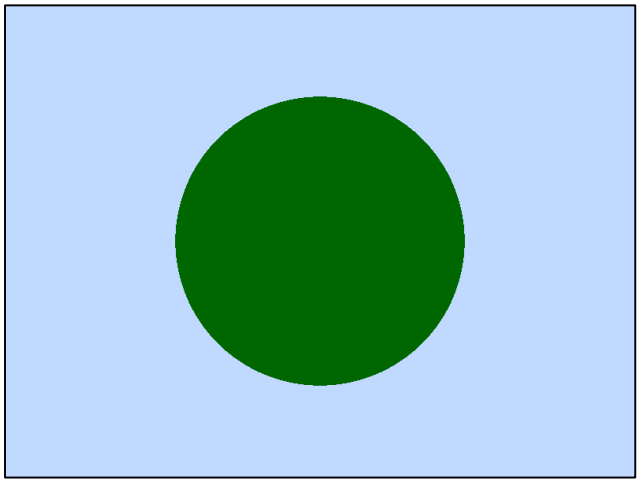
A sugárvetés (angolul: *ray casting*) egy rasztergrafikus képalkotó eljárás. A következő elven működik: Képzeljük el, hogy a világot egy szabályos, sűrű négyzetrácson keresztül nézzük (egyetlen szemünkkel, vagy akár kameránkkal). Az a feladatunk, hogy minden egyes négyzetet külön kiszínezzünk (egy négyzeten belül csak egyetlen színt használhatunk fel) úgy, hogy végül a négyzetrács egészére ránézve a lehető legélethűbb képet kapjuk a mögötte lévő világról.

Hogyan döntsük el, hogy egy adott négyzet milyen színű legyen? Alapból legyen minden négyzet halványkék. Az egyszerűsítés kedvéért tegyük fel, hogy abban a világban, amit megfigyelünk, csupán egyetlen objektum van: egy sötétzöld gömb. Ha egy adott négyzeten keresztül látjuk a gömb egy részét (tehát a gömb felületéről fény jut a szemünkbe), akkor a négyzetet színezzük ki sötétzölden, egyébként hagyjuk halványkéken. De egy gömb felületéről a tér szinte minden irányába indulnak fénysugarak, ezeknek csak egy kis része ér a szemünkbe. Sokkal gyorsabb, ha használjuk a 3-as számmal jelölt optikai alaptörvényt (lásd: 3.2.1): fordítsuk meg a fénysugarak irányát, indítsuk őket a szemünkből, minden egyes négyzet felé egyet-egyet (backward raytracing). Ha tehát a szemünkből, egy adott négyzet irányába indított sugár ütközik a térbeli gömbbel, akkor fessük zöldre a négyzetet.

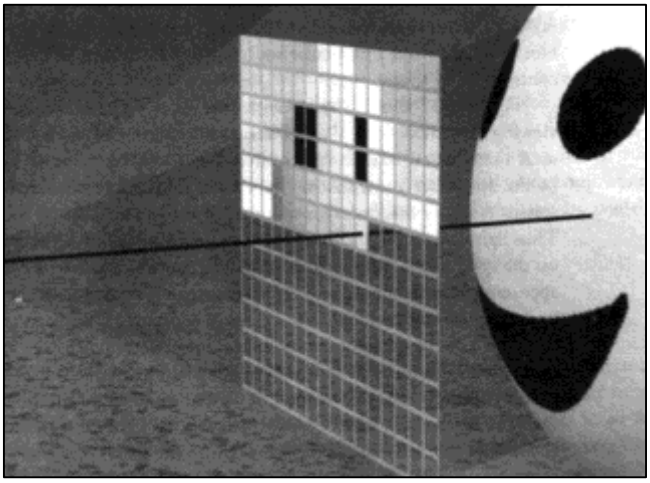


26. ábra. *Ray casting külső szemmel*

Mint láthatjuk az ábrákon, a megvalósított négyzetrács (a kamera szemögéből) sokkal sűrűbb, mint a külső szemlélős képeken, hiszen azok csak illusztrációk. Minél élesebb képet akarunk kapni eredményül, annál sűrűbbre kell vennünk a négyzetrácsot (növelnünk kell a felbontást). Ha elég nagyra állítjuk, akkor már emberi szemmel meg sem lehet különböztetni egymástól a pixeleket (persze ez a számítógépünk kijelzőjétől is függ). Ez természetesen növeli a program számításigényét is, tehát csökkenti a futásának a sebességét.



27. ábra. *Ray casting a kamera szemszögéből*



28. ábra. *Ray casting külső szemmel, bonyolultabb színekkel [11]*

Ez idáig a fő algoritmus csupán ennyiből áll (*pszeudokód*):

```
for each (Pixel p)
{
     ray.origin = eye;
     ray.direction = normalize(p.position – eye);
     if (intersectSphere(ray, sphere))
     {
          p.color = vec3(0.0, 0.4, 0.0);[10]
     }
     else
     {
          discard;[11]
     }
}
```

A szemből kilőtt sugarakat elsődleges sugaraknak (angolul: *primary rays*) hívjuk. Ha az objektummal való ütközés után, a metszéspontból további fénysugarakat indítunk (tükröződés, törés, vagy árnyék miatt), azt már sugárkövetésnek (angolul: *ray tracing*) nevezzük. Az eljárásnak léteznek olyan változatai is, amikor a fényforrásokból indítunk sugarakat (forward ray tracing, photon mapping, stb.), sőt van hibrid változat is (bidirectional ray tracing). Ezek képesek olyan vizuális jelenségek élethű megjelenítésére is, melyekre a klasszikus (backward) ray tracing esetén nem igazán van lehetőség (pl. *kausztika (caustic), subsurface scattering, diffuse interreflection*). Sokkal nagyobb számításigényük miatt azonban még pár évet/évtizedet biztosan várni kell, hogy a hardverek képesek legyenek ezeket valós időben futtatni.

3.4.2. Rekurzív sugárkövetés

Arthur Appel 1968-ban bemutatott ray casting algoritmusa után a következő nagy áttörést Turner Whitted érte el 1979-ben. Az ő ötlete volt az, hogy ne csak az első eltalált tárgyig kövessük a szemből kilőtt fénysugarakat, hanem folytassuk az útját. Három féle új fénysugarat indíthatunk az eltalált felületről, ezzel lehetővé téve tükröző és fénytörő felületek megjelenítését, illetve egészen triviális módon a vetített árnyékokat is. Utóbbihoz elegendő csupán a felület minden egyes pontjáról további fénysugarakat lőni a fényforrások felé. Ha ezek a fénysugarak eltalálnak eközben egy (átlátszatlan)

[10] A színeket RGB szerint, három elemű vektorban (vec3) adom meg úgy, hogy a lehetséges legkisebb érték a 0, a legnagyobb az 1.

[11] A *discard* kulcsszóval „eldobjuk” az adott pixelt, így a helyén az előre definiált, halványkék törlési szín lesz látható.

objektumot, akkor a felület pontja (ahonnan a sugarat lőttük) egyszerűen ne legyen megvilágítva: maradjon rajta az ambiens szín, ne adjunk hozzá diffúzt, se spekulárist (lásd: 3.6.1).

Ha egy felület tükröz, akkor legegyszerűbb a normálvektora alapján a tükörirányt kiszámolni (*reflect* függvény). A numerikus pontatlanság miatt a metszéspont egy egész picivel a testen belül is elhelyezkedhet. Ekkor az innen indított sugár beleütközhet még egyszer a test felületébe (belülről). Megoldás: toljuk el az új fénysugár kezdőpontját a normálvektor irányába (csak épphogy, epszilon mértékével). Ezt a „trükköt" alkalmazhatjuk a fényforrások felé indított sugarak esetén is (vetített árnyékok számításánál).

Ha áttetsző, és töri a fényt, akkor ugyancsak a normálvektor (és a relatív törésmutató) alapján számolhatjuk ki a törésirányt (*refract* függvény). Figyeljünk rá, hogy ha a fénysugár kifele jön az objektumból, akkor a felület normálvektorának ellentettjére kell a függvényt hívni (valamint az epszilonnal való eltolást is ennek megfelelően végezzük el).

Ha egy felület tükrözi, és töri is a fényt, akkor ezek különböző arányban jelennek meg az objektumon. Ezeket az arányokat a Fresnel törvények szerint, és a kioltási tényezők figyelembevételével programoztam le. Ez utóbbinak köszönhető az aranyszerű megjelenése az egyik gömbnek.[29]

A felhasznált, vektorokkal kapcsolatos, valamint trigonometrikus függvények jelentős része be van építve a GLSL-be (*dot, cross, normalize, reflect, reflact, length, distance, cos, sin, acos,* stb.)[27]. Ezek jellemzően hardveresen támogatott funkciók, így nagyon gyorsan, akár egyetlen órajel alatt képesek lefutni.

Az új sugarak indítását legegyszerűbben rekurzív módon lehet megvalósítani, azonban itt felmerül egy jelentős probléma: a GPU-k (jellemzően) nem tudnak olyan programot futtatni, amik rekurziót tartalmaznak.

3.4.3. Iteratív sugárkövetés

A GPU-k jelentős része nem képes függvényhívásokat végrehajtani, csak az újabbak, Nvidia CUDA technológiával. A függvényhívások azonban a mai napig sem hatékonyak a GPU-n: lassúak, korlátaik vannak. A GLSL-ben eleve tiltott a rekurzió

használata: ha a *compiler* rekurziót érzékel, rögtön hibát jelez. A látszólagos függvényhívás GLSL-ben (és jellemzően minden GPU-nyelven) csupán egy illúzió. A fordító minden egyes ciklust, függvényhívást, stb. kicsomagol, hogy végül csak a sorokat kelljen egymás után lefuttatnia. A rekurziót azonban nem tudja kicsomagolni, hiszen sokszor csak futásidőben derül ki, hogy hányszor kell lefuttatni. Akkor mégis hogyan lehet egy rekurzív sugárkövető algoritmust GLSL-ben megírni?

Ha egy probléma megoldható rekurzív algoritmussal, akkor megoldható iteratívval is. Ebből kiindulva biztos voltam benne, hogy nem én leszek az első, aki megpróbálkozik a rekurzív sugárkövetés iteratív megoldásának leprogramozásával. Az interneten való kutatásaim egy ilyen algoritmus után azonban rendre zsákutcába futottak, hiszen a talált iteratív ray tracer algoritmusok szinte mindig csak egyetlen irányba folytatták a sugár útját: eldöntötték egy objektumról, hogy az töri, vagy tükrözi a fényt, a kettőt együtt nem engedte meg. A *trace* függvény megoldásához végül Günther Voglsam diplomamunkáját használtam fel[28], az ebben lévő algoritmus állt legközelebb az én elképzeléseimhez. Ezt aztán módosítottam, és kiegészítettem a saját ötleteimmel a következő módon:

```
vec3 trace(in Ray ray)
{
  vec3 color = vec3(0.0);
  HitRec closestHit;
  float u,v;
  vec2 uv;
  Stack stack[STACK_SIZE]; // max depth
  int stackSize = 0;       // current depth
  int bounceCount = 1;
  vec3 coeff = vec3(1.0);
  bool continueLoop = true;
  while (continueLoop)
  {
    if(findClosest(ray, closestHit, bounceCount))
      {
        // Normalmap
        // …
        bounceCount++;
        Material mat = getMaterial(closestHit.ind);
        vec3 shadeCol = shade(closestHit, ray);
        color += shadeCol*coeff;
        // Textures
        // …
        if ((mat.reflective || mat.refractive) && bounceCount <= depth)
        {
```

```
        bool TIR = false;   // Total Internal Reflection
        if (mat.refractive) // Glass
        {
          float eta = 1.0/mat.n;
          Ray refractedRay;
          // Coming from outside the object  ?
          refractedRay.dir = dot(ray.dir, closestHit.normal) <= 0.0 ?
          refract(ray.dir, closestHit.normal, eta) :    // Yes
          refract(ray.dir, -closestHit.normal, 1.0/eta);// No
          TIR = length(refractedRay.dir) < EPSILON;
          if (TIR)
          {
            ray.dir = normalize(reflect(ray.dir, -closestHit.normal));
            ray.origin = closestHit.point - closestHit.normal*EPSILON;
          }
          else // Not Total Internal Reflection
          {
            refractedRay.origin = closestHit.point +
                                  closestHit.normal*EPSILON*
                                  sign(dot(ray.dir,closestHit.normal));
            refractedRay.dir = normalize(refractedRay.dir);
            if (!mat.reflective)
            {
              ray = refractedRay;
            }
            else
            {
              stack[stackSize].coeff = coeff*(vec3(1.0) -
              fresnel(refractedRay.dir, closestHit.normal, mat.f0));
              stack[stackSize].depth = bounceCount;
              stack[stackSize++].ray = refractedRay;
            }
          }
        }
        if ((mat.reflective && !TIR) // Mirror
        {
          if (dot(ray.dir, closestHit.normal) < 0.0)
          {
            coeff = coeff*fresnel(ray.dir, closestHit.normal, mat.f0);
            ray.dir = normalize(reflect(ray.dir, closestHit.normal));
            ray.origin = closestHit.point + closestHit.normal*EPSILON;
          }
          else
          {
            continueLoop = false;
          }
        }
      }
      else // Diffuse material
```

```
        {
          continueLoop = false;
        }
      }
    else // No hit
    {
      color += vec3(0.6, 0.75, 0.9)*coeff;
      continueLoop = false;
    }
    // Glow effect around the sun
    // …
    if (!continueLoop && stackSize > 0)
    {
      ray = stack[--stackSize].ray;
      bounceCount = stack[stackSize].depth;
      coeff = stack[stackSize].coeff;
      continueLoop = true;
    }
  }

  return color;
}
```

Az algoritmus lényegi része, hogy egy cikluson belül minden egyes olyan esetben, amikor a fénysugár kétfelé mehet tovább (tükör-, és törésirány), az egyik sugarat beletesszük egy feldolgozandó verembe (*stack*). Az algoritmus csak akkor áll le, ha üres a verem, nincs több feldolgozandó sugár.

3.5. A metszőfüggvények

A metszéspontkeresés a sugárkövetésnek az egyik kulcsfontosságú, jellemzően a legtöbb időt igénylő része. Segítségével egy egyenes (fénysugár) és egy geometriai alakzat legközelebbi (kamerával szemben lévő) metszéspontját számoljuk ki (ha van ilyen). A sugárnak (Ray) kezdőpozíciója (*origin*), és (normalizált) iránya (*dir*) van, ezeket háromelemű vektorokkal ábrázoljuk. A *t* változó minden metszőfüggvényben az eltalált (legközelebbi) pont, és a sugár kezdőpozíciója közti távolság. A kapott eredményeket (ütközési pont, normál vektor, stb.) egy *ütközési rekord*ba tesszük (*HitRec*). Ennek fontos szerepe van a *trace()* (lásd: 3.4.3), és a *shade()* (lásd: 3.6.1) függvényekben, valamint textúrázáskor (lásd: 3.6.2). A jobb olvashatóság miatt a megvalósított függvényeket ide paraméterek nélkül írom le.

Ha több objektummal is keresünk metszéspontot a színtérben, akkor ezek közül mindig csak a legközelebbit jelenítsük meg. Ez az eljárás az én programomban egy egyszerű *minimumkeresés*sel van megoldva a *findClosest()* függvényben.

3.5.1. Az intersectSphere() függvény

Gömb esetében a metszéspontok megtalálásához egy másodfokú egyenletet kell megoldani (egy egyenes legfeljebb kétszer metszhet egy gömböt).

A gömböt egy négyelemű vektor reprezentálja: első három eleme a középpontjának *x, y, z* koordinátája, negyedik a sugara. Ezeket figyelembe véve az egyenlet megoldásának egyik lehetséges algoritmusa a következő[1]:

```
bool intersectSphere()
{
      vec3 dist = ray.origin - sphere.xyz;
      float b = dot(dist, ray.dir)*2.0;
      float a = dot(ray.dir, ray.dir);
      float c = dot(dist, dist) - sphere.w*sphere.w;
      float discr = b*b - 4.0 * a * c;
      if (discr < 0.0) return false;
      float sqrtDiscr = sqrt(discr);
      float t1 = (-b + sqrtDiscr)/2.0/a;
      float t2 = (-b - sqrtDiscr)/2.0/a;
      float t;
      if (t1 < EPSILON) t1 = -EPSILON;
      if (t2 < EPSILON) t2 = -EPSILON;
      if (t1 < 0.0) return false;
      if (t2 > 0.0) t = t2;
```

```
    else t = t1;
    hitRec.t = t;
    hitRec.origo = vec3(sphere.xyz);
    hitRec.point = ray.origin + t*ray.dir;
    hitRec.normal = normalize(hitRec.point - hitRec.origo);

    return true;
}
```

Itt nyomon követhetjük a különböző eseteket:

- Amennyiben a diszkrimináns negatív, nincs (valós) megoldás, a szemből „kilőtt” egyenes nem találja el a gömböt, nincs metszéspont, *hamis*sal térünk vissza.
- Ha csak negatív megoldás van, akkor a gömb a kamera mögött helyezkedik el, tehát ismét hamissal térünk vissza.
- Ha az egyik megoldás pozitív, a másik negatív, akkor a kamera a gömb belsejében van: tároljuk el a gömb belső (kamerával szemben lévő) falán lévő metszéspontot, és az ahhoz tartozó egyéb fontos adatokat.
- Ha két pozitív megoldás van: a szemünkből „kilőtt” egyenes kétszer metszi a gömböt (egyszer bemegy rajta, egyszer kijön). Csak a közelebbi metszéspontot (és a hozzá tartozó adatokat) tároljuk el.

3.5.2. Az intersectPlane() függvény

A síkok metszése talán a legkönnyebb metszésfeladat, mind a számítógép, mind a programozó számára. A *t* változó kiszámítása a következő gondolatmenet alapján történik:

- Egy *p* pont akkor van rajta a síkon, ha *dot(n, p - q) = 0*, ahol *dot* a skaláris szorzat (*dot product*), *n* a sík normálvektora, *q* pedig a síkon egy pont (ez a két vektor elegendő egy sík egyértelmű definiálásához). Az egyenlet lényegében azt fejezi ki, hogy a *q* és *p* által meghatározott egyenes merőleges-e a sík normálvektorára. (Ha igen, *p* rajta van a síkon.)
- A *p* pontot a következő összefüggéssel is felírhatjuk: *p = ray.origin + t*ray.dir*. Ezt behelyettesítve a következő egyenletet kapjuk: *dot(n, ray.origin+t*ray.dir-q) = 0.*
- Innen *t* változót kifejezve meg is kaphatjuk az algoritmusban szereplő kifejezést.

Az algoritmus[13]:

```
bool intersectPlane()
{
	float t = dot(plane.n,(plane.q-ray.origin)) / dot(plane.n,ray.dir);
	if (t < EPSILON ) return false;

	hitRec.t = t;
	hitRec.origo = plane.q;
	hitRec.point = ray.origin + t*ray.dir;
	hitRec.normal = plane.n;

	return true;
}
```

3.5.3. Az intersectDisc() függvény

A korong metszéséhez felhasználjuk a sík metszését is. Miután a korong síkját metszi a fénysugár, el kell dönteni, hogy az eltalált pont a korong közepétől egy bizonyos távolságon (*disc.r* = a korong sugara) belül van-e. Ha igen, akkor van metszés a koronggal (*true*), különben nincs (*false*).

A gondolatmenet segítségével szinte bármilyen síkidomot metszhetünk a megfelelő feltétel matematikai megfogalmazásával. A korong metszését a következő algoritmussal implementáltam[13]:

```
bool intersectDisc()
{
	Plane plane;
	plane.n = disc.n;
	plane.q = disc.o;
	if (intersectPlane(ray, plane, hitRec))
	{
		vec3 p = ray.origin + hitRec.t*ray.dir;
		if (distance(p, disc.o) <= disc.r)
		{
			return true;
		}
	}
	return false;
}
```

3.5.4. Az intersectTriangle() függvény

A háromszögek metszését az alkotóiról elnevezett Möller-Trumbore algoritmus [14] alapján számítom ki:

```
bool intersectTriangle()
{
	vec3 e1, e2;  //Edge1, Edge2
	vec3 P, Q, T;
	float det, invDet, u, v;
	float t1;
	//Find vectors for two edges sharing V1
	e1 = t.B - t.A;
	e2 = t.C - t.A;
	P = cross(ray.dir, e2);
	det = dot(e1, P);
	invDet = 1.0 / det;
	T = ray.origin - t.A;
	u = dot(T, P) * invDet;
	//The intersection lies outside of the triangle
	if(u < 0.0 || u > 1.0) return false;
	//Prepare to test v parameter
	Q = cross(T, e1);
	//Calculate V parameter and test bound
	v = dot(ray.dir, Q) * invDet;
	//The intersection lies outside of the triangle
	if(v < 0.0 || u + v  > 1.0) return false;
	t1 = dot(e2, Q) * invDet;
	if(t1 > EPSILON) //ray intersection
	{
		hitRec.t = t1;
		hitRec.point = ray.origin + ray.dir * t1;
		hitRec.normal = normalize(cross(t.B-t.A, t.C-t.A));
		hitRec.origo = (t.A+t.B+t.C)/3.0;

		return true;
	}

	return false; // No hit, no win
}
```

Természetesen onnantól kezdve, hogy háromszögeket ki tudunk rajzolni, ezek sokaságából (hasonlóan az inkrementális képszíntézishez) bármilyen alakzatot meg tudunk jeleníteni, de ebben a programban ez a lehetőség csak minimálisan van használva, hiszen nem ez a célja.

3.5.5. Az intersectTorus() függvény

A tórusz metszéséhez negyedfokú egyenletet kell megoldani (egy egyenesnek és egy tórusznak legfeljebb négy közös pontja lehet). Ennek megvalósításához Íñigo Quílez grafikus-programozó-művész algoritmusát[15] használtam fel:

```
bool intersectTorus()
{
     ray.origin.x -= 10.0;
     float Ra2 = torus.x*torus.x;
     float ra2 = torus.y*torus.y;
     float m = dot(ray.origin, ray.origin);
     float n = dot(ray.origin, ray.dir);
     float k = (m - ra2 - Ra2)/2.0;
     float a = n;
     float b = n*n + Ra2*ray.dir.z*ray.dir.z + k;
     float c = k*n + Ra2*ray.origin.z*ray.dir.z;
     float d = k*k + Ra2*ray.origin.z*ray.origin.z - Ra2*ra2;
     float p = -3.0*a*a     + 2.0*b;
     float q =  2.0*a*a*a   - 2.0*a*b   + 2.0*c;
     float r = -3.0*a*a*a*a + 4.0*a*a*b - 8.0*a*c + 4.0*d;
     p /= 3.0;
     r /= 3.0;
     float Q = p*p + r;
     float R = 3.0*r*p - p*p*p - q*q;
     float h = R*R - Q*Q*Q;
     float z = 0.0;
     if( h < 0.0 )
     {
          float sQ = sqrt(Q);
          z = 2.0*sQ*cos( acos((R/(sQ*Q))) / 3.0 );
     }
     else
     {
          float sQ = pow( sqrt(h) + abs(R), 1.0/3.0 );
          z = sign(R)*abs( sQ + Q/sQ );
     }
     z = p - z;
     float d1 = z   - 3.0*p;
     float d2 = z*z - 3.0*r;
     if( abs(d1)<EPSILON )
     {
          if( d2<0.0 ) return false;
          d2 = sqrt(d2);
     }
     else
     {
          if( d1<0.0 ) return false;
```

```
            d1 = sqrt( d1/2.0 );
            d2 = q/d1;
        }
        float result = 1e20;
        h = d1*d1 - z + d2;
        if( h>0.0 )
        {
            h = sqrt(h);
            float t1 = -d1 - h - a;
            float t2 = -d1 + h - a;
            if( t1>0.0 )      result=t1;
            else if( t2>0.0 ) result=t2;
        }
        h = d1*d1 - z - d2;
        if( h>0.0 )
        {
            h = sqrt(h);
            float t1 = d1 - h - a;
            float t2 = d1 + h - a;
            if( t1>0.0 )      result=min(result,t1);
            else if( t2>0.0 ) result=min(result,t2);
        }
        if (result > 0.0 && result < 100.0) //hit
        {
            hitRec.t = result;
            hitRec.point  = ray.origin + hitRec.t*ray.dir;
            hitRec.normal = normalize(hitRec.point *
                            (dot(hitRec.point, hitRec.point) -
                            torus.y*torus.y - torus.x*torus.x*
                            vec3(1.0,1.0,-1.0)));
            return true;
        }

        return false; //no hit
}
```

3.6. Az objektumok anyagtulajdonságai

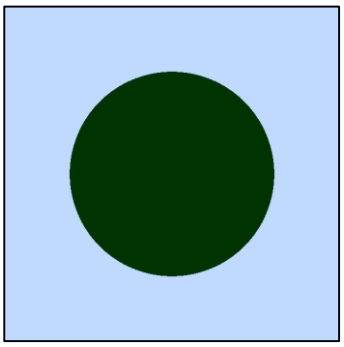
Ebben a fejezetben bemutatom, hogy a programomban a tárgyak anyagi jellemzői milyen módszerek, technológiák alapján valósul meg.

3.6.1. A Blinn-Phong árnyalás

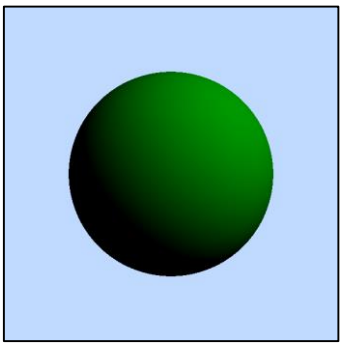
A *ray casting*nál (lásd: 3.4.1) láthattuk, hogy a megjelenített gömb csupán elsődleges sugarakkal metszve csak egy korongnak néz ki, hiszen minden kirajzolt pontját ugyanolyan (sötétzöld) színűre festettük. Nem feltétlenül szükséges azonban további fénysugarakat indítani ahhoz, hogy térhatást érjünk el.

Az árnyalások (angolul: *shading*) olyan eljárások, melyek különböző színárnyalatok használatával növelhetik egy jellemzően 3D-s modellben a mélységérzetet. A 70-es években ennek több fajta megvalósítása is született; egyik legismertebb, és legnépszerűbb az alkotóiról elnevezett Blinn-Phong árnyalás. Ez (sok másikhoz hasonló módon) három összetevőből áll:

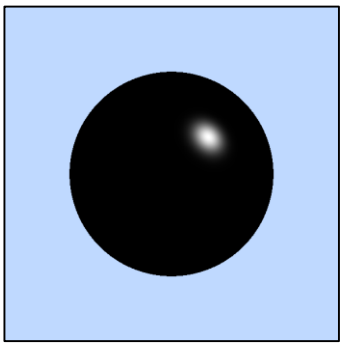
1. Ambiens komponens: Ilyennek látunk minden olyan tárgyat, melyet közvetlenül nem ér fény.
2. Diffúz komponens: Önmagában (spekuláris nélkül) matt felületek megjelenítésénél használják. Ahol nagy a fénysugár és a felület síkja által bezárt szög, ott viszonylag erőteljes. Ahogy csökken a szög, úgy csökken az intenzitás is (a sinus függvény szerint).
3. Spekuláris komponens: Csillogó felületeken apró, (általában) fehér, fényes foltként jelenik meg azokon a területeken, ahol a fényforrásból jövő sugarat megtükrözve olyan sugarat kapunk, amely a szemünk felé mutat. A tükrözött sugárnak elég csupán az irányát meghatározni, nem kell új sugarat „lőni”.



29. ábra. *Ambiens komponens*

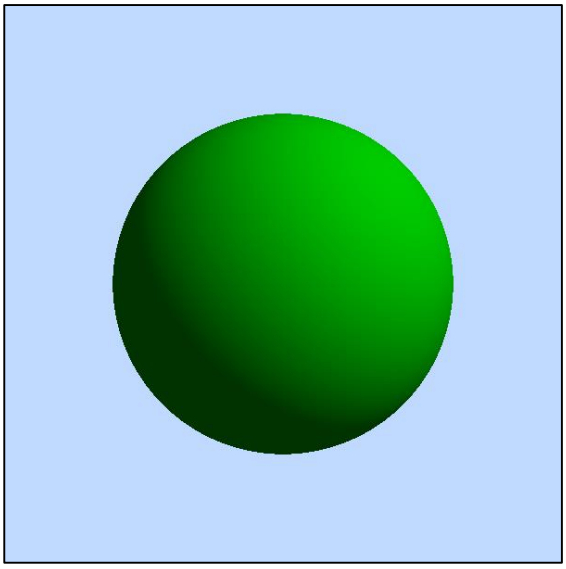


30. ábra. *Diffúz komponens*

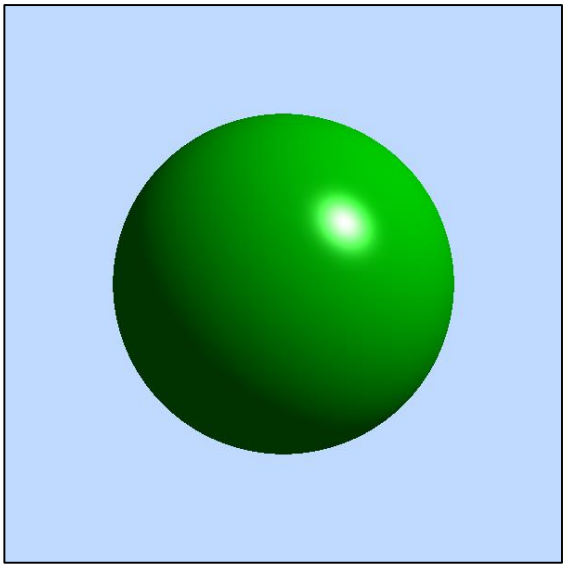


31. ábra. *Spekuláris komponens*

Ezek összege matt, vagy csillogó felületű modellt eredményez:



32. ábra. *Ambiens + diffúz*



33. ábra. *Ambiens + diffúz + spekuláris*

Az árnyaláshoz természetesen definiálnunk kell a térben egy fényforrást. Ennek két legegyszerűbb módja, ha pontszerű, vagy irány-fényforrást adunk meg, mivel mindkettőhöz elég egy háromelemű vektor (*vec3*). A képeken a gömb az origóban van, sugara 1 egység, a fény pozíciója pedig a (10, 10, 10) koordinátájú pont. Figyelembe véve, hogy alapesetben az értékek lehetnének néhol negatívak is, és ez bizonyos szemszögből látványbeli hibát okozhat; a Blinn-Phong árnyalást a következőképpen implementáltam:

```
vec3 shade(in HitRec closestHit, in Ray ray)
{
  vec3 refDir   = normalize(reflect(closestHit.point - ray.origin,
                  closestHit.normal));
  vec3 diffuse  = vec3(0.0);
  vec3 specular = vec3(0.0);
  vec3 toLight  = normalize(lightPos - closestHit.point);
  float diffintensity = clamp(dot(closestHit.normal, toLight), 0.0, 1.0);
  diffuse = clamp((dif*diffintensity), 0.0, 1.0);
  specular = clamp((spec*pow(clamp(dot(toLight, refDir),
                             0.0, 1.0), 30.0)), 0.0, 1.0);
  vec3 color = amb + diffuse + specular;
  return color;
}
```

A spekuláris komponens számításánál a kitevő azt befolyásolja, hogy a csillanás mekkora felületen legyen (minél nagyobb a kitevő, annál kisebb a felület).

Az *amb*, *dif*, és *spec* változók egy adott anyagra (material) jellemző tulajdonságok, így a későbbiekben, ha már sok objektum van a színtérben, ezeket érdemes egy tömbben tárolni, majd ebből visszaadni az éppen megfelelőt. A GLSL sajátosságai miatt azonban minden egyes materialt külön változóként implementáltam (material0, material1, …).

A komponenseket (az ambiens kivételével) minden egyes fényforrásra ki kell számolnunk, így több fényforrás esetén ezeket célszerű egy ciklusban végrehajtani. A vetített árnyékokat is érdemes itt, a *shade* függvényben megvalósítani (lásd: 3.4.2).

3.6.2. A textúrák, a skybox

A textúrák digitális képek, melyeket egy mintavételező (angolul: *sampler*) segítségével ráfeszíthetünk egy felületre. Sík, tengelyekkel párhuzamos alakzatokra szinte triviális ennek a legprogramozása. A fő probléma abból szokott adódni, hogy egy síkbeli képet miként helyezünk fel egy térbeli, akár görbe felületre. Ezt a folyamatot *UV mapping*nek nevezzük. Munkám során csak a gömbök felületére kellett UV mapping megoldást keresnem[16], az összes többi textúra tengelyekkel párhuzamos sík felületekre van feszítve, ebbe beleértve a hátteret is (skybox), ami igazából 6db kép, 6db síkon. Az ezek által meghatározott kocka belsejében helyezkedik el minden más objektum. Mivel a síkok az origótól 10 000 egység távolságra vannak (viszonyításképpen: a tér elemei bőven belefférnek egy 100 egység oldalú kockába), így azt az érzést keltik a felhasználóban, hogy ez valójában egy egységes háromdimenziós háttér, hiszen ilyen távolságból, a megfelelően összeillesztett síkokat lehetetlen megkülönböztetni egymástól.

3.6.3. A buckatérképek

A buckatérképek (*bump maps*) lényegében speciális textúrák. Ezeket nem színként jelenítjük meg a felületeken, mint a hagyományos textúrákat, hanem segítségükkel göcsörtös kinézetet adhatunk egy sima felületnek. A *bump mapping* James Blinn fejében született meg[17], aki az egyik alkotója a már emlegetett Blinn-Phong shadingnek. Számos megvalósítása létezik, legelterjedtebb, és legismertebb az ún. *normal mapping*.

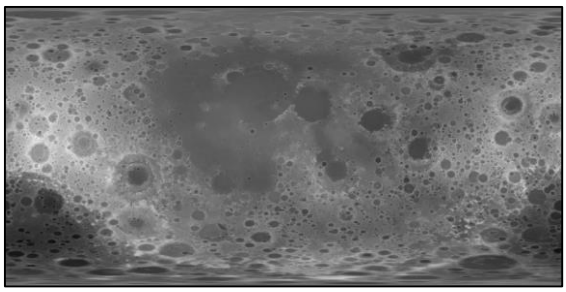
A normal mapping segítségével egy adott felület normálvektorait változtatjuk meg. Ez elegendő ahhoz, hogy az egyenletes, sima felület egyenetlennek, göcsörtösnek tűnjön. Működésének alapja, hogy egy adott pixeléből kiolvasott színértékek 1-1 vektornak felelnek meg. Egy normalmap jellemzően kékes színű, mivel az RGB szerinti (0.5, 0.5, 1.0) szín jelenti az „alap" normálvektort, tehát ekkor nem változtatjuk egyik irányba sem. Az ettől való eltéréseket kell valamilyen módon láthatóvá tennünk.

Az én megvalósításomban a normal mapping lépései a következők:

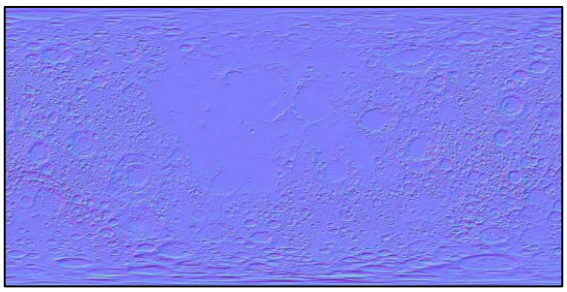
1. Sampler segítségével a normálvektor kiolvasása a normalmapből, redukálása, és normalizálása, hogy pl.: a (0.5, 0.5, 1.0) vektorból (0.0, 0.0, 1.0) legyen: `normalFromMap=normalize(2.0*((texture(earthNormalMap,uv)).rgb)-1.0);`[12]
2. Létrehozunk egy *R* nevű, ún. *rotációs* mátrixot. Ennek megvalósításához felhasználtam egy Neil Mendoza nevű grafikus-művész által közzétett függvényt.[18] A rotációs mátrix lényegében leírja, hogy egy adott pontban (ha az az XY síkban lenne) lévő (0.0, 0.0, 1.0) irányú normálvektor mennyivel tér el a kiolvasott normálvektortól. Tehát ha ezt a rotációs mátrixot beszorozzuk a (0.0, 0.0, 1.0) vektorral, akkor megkapjuk egy adott pontra a térbeli normálvektort.
3. Szorozzuk be R-et a *sampler*rel kiolvasott normálvektorral, és legyen ez az új normálvektor.
4. Ismételjük meg az első három lépést az objektum összes normálvektorára.

[12] Normalizálni ideális esetben nem is kellene, de jobbnak láttam benne hagyni, hogy ezzel kiküszöböljük az esetleges rosszminőségű normalmapekből adódó hibákat.

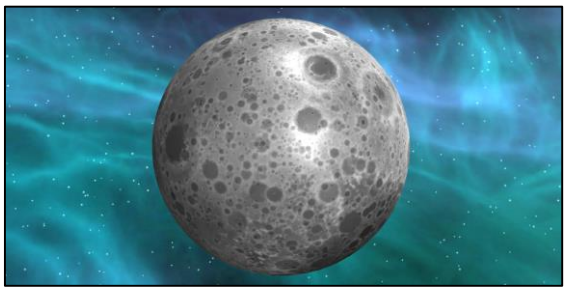
A felhasznált textúrákat, buckatérképeket többnyire a Google keresővel találtam.[19][20] A skybox képeit egy iskolatársam projektjéből töltöttem le, ő már nem tudta megmondani, hogy honnan vannak.[21] Némelyik buckatérképet pedig textúrából generáltattam egy online eszközzel.[22]



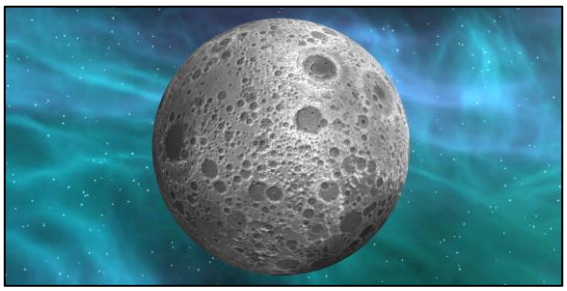
34. ábra. *A „Hold” textúrája*



35. ábra. *A „Hold” normalmapje*



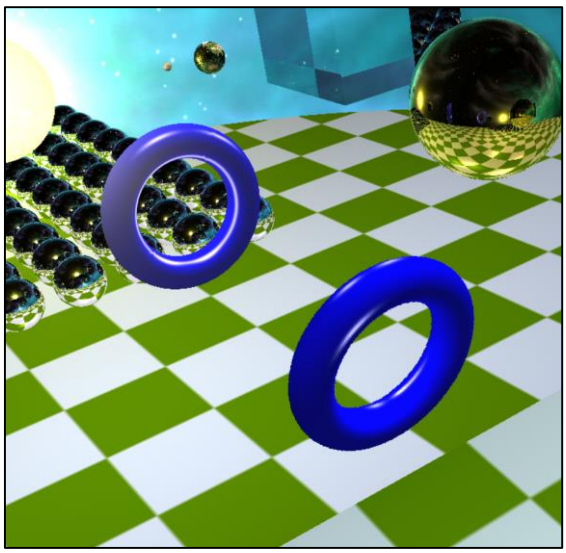
36. ábra. *A „Hold” normalmap nélkül*



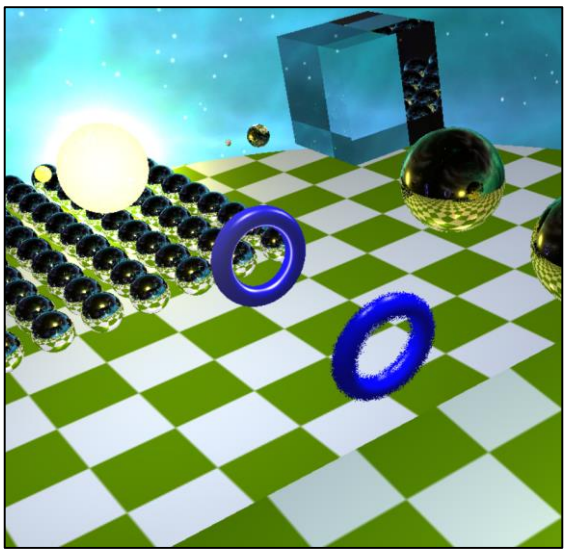
37. ábra. *A „Hold” normalmappel*

3.7. Tórusszal kapcsolatos problémák, és megoldások

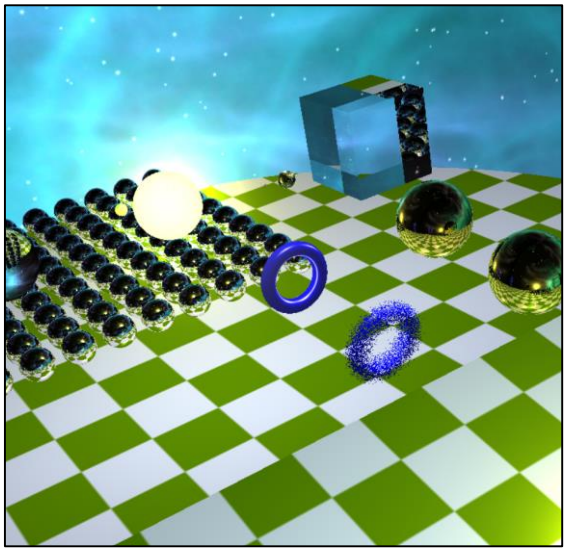
Kezdetben a megjelenített tórusz formája a nézőpont (kamera) és a tórusz közötti távolság növekedésével drasztikusan romlott:



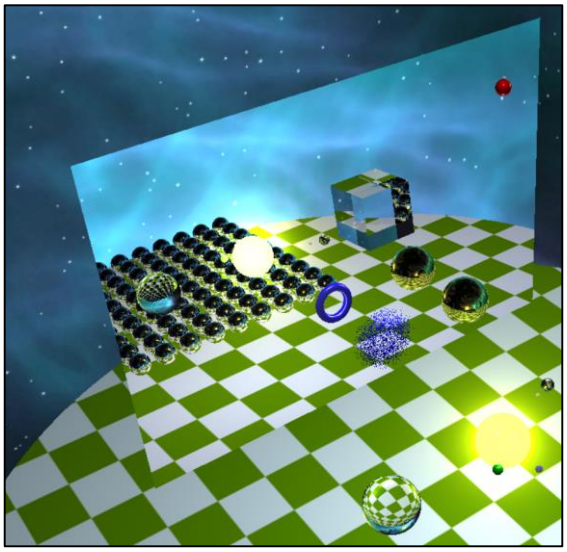
38. ábra. *Tórusz, és tükörképe közelről*



39. ábra. *Tórusz, és tükörképe messzebbről*



40. ábra. *Tórusz, és tükörképe még messzebbről*



41. ábra. *Tórusz, és tükörképe messziről*

A teszteléseim alapján valószínűsíthető lett, hogy a jelenség oka visszavezethető a *float* típus pontatlanságára. A tórusz metszése komplex feladat, negyedfokú egyenletet kell megoldani hozzá (lásd: 3.5.5). A megoldáshoz trigonometrikus, és exponenciális függvények használatára is szükség van, azonban némi próbálgatás, és utánajárás után kiderült, hogy a GLSL nem támogatja *double* típussal ezeket a függvényeket, (és mint kiderült, sokkal nehezebb implementálni őket annál, mint ahogy azt elsőre gondoltam). A következő ötletem volt a kérdéses függvények közelítése Taylor-sorokkal, azonban ezt

az utat is hamar elvetettem, hiszen jelentősen romlott a program hatékonysága, és pontosabb sem lett tőle a metszéspontkeresés; így a float típusok double-re cserélése, mint lehetőség, zsákutcába futott. További lehetőség volt, hogy háromszögek sokaságából rakom össze a tóruszt, azonban én (a ray tracing előnyeit kihasználva) mindenképpen analitikus megoldást szerettem volna találni.

A pontatlan tórusz további problémát okozott a vetített árnyékok esetében is: a felületekről küldött fénysugarak olyan helyeken is metszést találtak (a tórusszal), melyeknek objektum még a közelében sem volt, így véletlenszerűnek tűnő pontokban is besötétültek az anyagok felszínei, ezzel szemcsés, láthatóan hibás képet eredményezve.

Az ábrákon jól látható, hogy a tükörképben megjelent tórusz a nagy távolság ellenére is pontos marad, megtartja a formáját. Ennek oka, hogy a tükör felszínéről indított sugár kezdőpozíciója, és a tórusz közötti távolság konstans (és kicsi), nem nő attól, hogy mi a kameránkkal messzebbre megyünk. Ezt használtam ki végül a megoldásomban: a tóruszt beletettem egy üvegkockába. A kocka felszínéről (megtört irányban) továbbküldött sugár „közelről” metszi a tóruszt, ezzel elkerülve a vele kapcsolatos problémák nagyobb részét. Az árnyékokkal kapcsolatos problémára azt a megoldást találtam ki, hogy a tóruszt eleve csak akkor keressük, ha a metszésben résztvevő sugár mélysége legalább 2 (tehát közvetlenül a szemünkből jövő sugárral NE keressünk metszéspontot, csak ha tükörben, vagy üvegen át látszana a tórusz). Ezek a szaknyelv szerint persze jobban hasonlítanak *kerülő megoldások*ra (angolul: *workaround*), mint *igazi megoldások*ra (angolul: *solution*). Annyi „csalás” még van a dologban, hogy a tóruszon megjelenő árnyalás nem veszi figyelembe azt, hogy körülötte üveg van (nem törnek a fények felé mutató sugarak), sőt az egész tórusz pozíciójának eltolása egy illúzió: valójában a tóruszt metsző sugarakat tolom el ellentétes irányba, emiatt a tóruszon minden árnyalás úgy működik, mintha az az origóban lenne (ez természetesen csak akkor tűnhet fel, ha részletesen megvizsgáljuk).

3.8. A V-sync

Sokan találkozhattunk már az ún. *screen tearing* jelenséggel. Ez akkor fordul elő, amikor a számítógépünk monitorja egyszerre egynél több képkocka (*frame*) tartalmát is megpróbálja megjeleníteni.

Ez például úgy történhet meg, hogy a monitor akkor mutat meg egy új képkockát, amikor az még nincsen kész. Ekkor a befejezetlen rész még az előző képkockából való. Ezt általában egy vízszintes irányú „törés” jelzi:

42. ábra. *Pillanatkép a Counter-Strike: Global Offensive című videójátékból (illusztráció)*

Ezt a jelenséget hívatott kijavítani a *vertikális szinkronizáció*, vagy röviden *V-sync*. Ennek röviden az a lényege, hogy a GPU nem kap engedélyt arra, hogy bármi látható dolgot műveljen a kijelző memóriáján mindaddig, amíg a monitor be nem fejezi a frissítési ciklusát.[12]

A V-syncnek többféle megvalósítása létezik: egyik legnépszerűbb, és legegyszerűbb az ún. dupla-pufferelés (angolul: *double-buffering*), melynek előnye, hogy alkalmazásával eltűnik a frusztráló *screen tearing* effektus, és nem engedi a GPU-t a rákötött kijelző frissítési rátájánál (LCD-knél jellemzően 60 FPS) gyorsabban rajzolni (ezért fut le maximum 60-szor a render() függvény), ezzel adott esetben energiát is spórol.

Hátránya, hogy emiatt egy minimális *beviteli késleltetést* (angolul: *input lag*) érzékelhetünk, így ahol fontos, hogy nagyon gyorsan reagáljunk valamire (jellemzően FPS-videójátékok esetében), ott általában ki szokták kapcsolni. További hátránya, hogy

amint a GPU által másodpercenként kirajzolt képkockák száma a kijelző frissítési rátája alá esik (pl.: 60 helyett csak 59-et rajzol a GPU), a megjelenített képkockák száma feleződik, hiszen minden képkockának éppen csak egy kicsivel több időre lenne szüksége ahhoz, hogy kész legyen, emiatt pedig mindig meg kell várni a következőt is (így az 59 FPS helyett effektíve csak 30 lesz). Bizonyos GPU-k biztosítanak olyan lehetőségeket, mint pl.: az *Adaptív V-sync*, melynek lényege, hogy csak 60 FPS fölött kapcsol be, ezzel kiküszöbölve az FPS-feleződést.

A legújabb megoldások, a *G-Sync (Nvidia)*, valamint a *FreeSync (AMD)* pedig a kijelző frissítési rátáját igazítják a GPU-hoz, ezzel biztosítva *tear*-mentes, folyamatos, szép képet. Ezt a technológiát természetesen a kijelzőnek is támogatnia kell, és sajnos az ilyen monitorok (és GPU-k) jelenleg még viszonylag drágák.

Összegezve tehát, ha bekapcsoljuk a V-syncet, akkor legfeljebb annyi FPS-ünk lesz, amennyi a monitorunk frissítési rátája, és minden egyes képkocka egységesen lesz kirajzolva. Ha azonban tudni akarjuk, hogy a GPU-nk valójában mire képes, akkor kapcsoljuk ki. Sok esetben akár többszázat is számlálhat az FPS-mérő.

3.8.1. Miben más a WebGL verzió

A webes verziók újonnan bevezetett sajátossága, hogy a *render*elés nem egy (ilyen helyzetben megszokott) végtelen ciklusban történik, hanem sokkal ajánlottabb az ún. *requestAnimationFrame()* függvényt használni, majd ezen belül saját magát (rekurzív módon) hívni (*callback*). Ezt Paul Irish *front-end* mérnök mutatta be 2011-ben. Állítása szerint ez főképp az alábbiak miatt jobb megközelítés, mint a hagyományos ciklusos megoldás[35]:

- A böngészők implementációja szerint optimalizálható, így folyékonyabb, és hatékonyabb képmozgást tehet lehetővé.
- Az inaktív *tab*okon megszűnik a mozgás, így kevesebb GPU-, CPU-, és memóriahasználat jellemző.
- Emiatt energiatakarékosabb is.

Nyilvánvaló módon emiatt a webes verziókban a V-sync mindig be van kapcsolva.

3.9. A heisenbug

A munkám vége felé, mikor a család, barátok, stb. számítógépein tesztelgettem a programot, furcsa hibára (angolul: *bug*) lettem figyelmes: viszonylag új Nvidia videókártyával rendelkező asztali számítógépek hibásan jelenítették meg a színteret; a tórusz egyáltalán nem jelent meg, néhány gömb közvetlenül megjelent, de a tükörképekben már nem, és hasonlók. A jelenség csak ezekkel az asztali számítógépekkel, csak Windows alatt, és csak az OpenGL verzióknál volt jelen, a webes verziók tökéletesen működtek, Linuxon a natívak is, régebbi Nvidia asztali videókártyával, valamint új Nvidia videókártyás laptoppal Windowson sem volt probléma. Napokig, vagy talán hetekig próbáltam rájönni a hiba okára, ám a GLSL-t nem olyan egyszerű debuggolni, mint ahogy azt egy CPU-n futó kódnál megszokhattuk: nincs rá beépített debugger, és adatot sem tudunk kiírni vele, maximum az egyes pixelek színét változtathatjuk. Végül addig kutakodtam az interneten a *shader debugging* lehetőségeiről, míg találtam egy *Nvidia Nsight* nevezetű bővítményt a Visual Studio fejlesztői környezethez. Ennek helyes használatának megértéséhez ezek a legfontosabb lépések:

- Program indítása a fejlesztői környezetből az Nsight bővítmény *Start Graphics Debugging* nevű opciójával.
- Amíg fut a program, válasszuk az Nsight menüjéből a *Pause and Capture Frame* opciót, ezzel szüneteljük a program futását, és az ablak következő (még ki nem rajzolt) képkockát mutatja.
- A még nem feldolgozott pixelek színe piros. Válasszunk ezek közül egy tetszőlegeset (amire debuggolni szeretnénk a fragmens shadert), majd nyomogassuk a „next” lehetőséget, amivel tovább léptetjük a programot. Látni fogjuk, ahogy véletlenszerűnek tűnő sorrendben a piros pixelek kiszíneződnek a megfelelő színekre. Ha az általunk választott pixel következik, azt látni fogjuk, hiszen az előre meghatározott *breakpoint*nál várni fog az utasításainkra.
- Innentől kezdve kb. úgy működik, mint egy megszokott debugger: Tudunk léptetni, kiolvasni a változók aktuális értékét, stb.

Betöltöttem tehát a projektet egy olyan számítógépen, amelyiken a hiba jelen volt, és indítottam az Nsight Graphics Debuggert. Nagy meglepetést okozott, mert ezzel az opcióval tökéletesen futott ezeken a rendszereken is, így gyakorlatilag a szó ma

megszokott értelmében a probléma nem volt debugolható, hiszen a debugolást elkezdve megszűnt maga a probléma. Ekkor tudatosult, hogy egy ún. *heisenbug*gal álltam szemben.

A heisenbug kifejezést Werner Heisenbergről[13], a kvantummechanika egyik úttörő fizikusáról nevezték el. A Wikipédián található definíció szerint ez egy olyan szoftverbug, amely eltűnik, vagy megváltoztatja viselkedését, ha valaki tanulmányozni próbálja.[30]

Még küzdöttem pár napot a problémával, de szorított az idő, így végül belenyugodtam abba, hogy néhány rendszeren a program működése kicsit hibás lesz. A megoldást az optimalizáció során, egészen véletlenül találtam meg.

[13] Heisenberg fogalmazta meg az ún. *megfigyelési effektus*t (angolul: *observer effect*), mely szerint egy rendszer megfigyelése elkerülhetetlenül megváltoztatja annak állapotát.

3.10. Optimalizáció

A rengeteg uniform változó átadása minden képkocka kirajzolásakor megterhelte a gépeket (különösen a webes verzió esetén), hosszú lett a fordítási idő, majd futáskor szaggatott a kép. Ezen elsősorban úgy javítottam, hogy a konstans értékek többségét (mint amilyenek a fényforrások, a materialok, nem mozgó objektumok, stb.) magában a fragmens shaderben definiáltam ahelyett, hogy továbbra is uniform változóként adtam volna át őket. Ez jelentősen javított a teljesítményen, azonban még egyáltalán nem voltam elégedett a webes verziók fordítási idejével. A „pro" verzió esetében ez annyira elhúzodott, hogy Windowson akár 40-50 másodpercet is várni kellett ahhoz, hogy képet lássunk (Linuxon jellemzően sokkal hamarabb készen volt). Ezt a legtöbb böngésző nem tolerálta, és még a betöltés előtt megszakította a videókártyával a kapcsolatot (lásd: 2.1.2).

Google-ben rákeresve találtam egy (online is elérhető) automatikus GLSL optimalizáló szoftvert.[31] Használata nagyon egyszerű: adott egy szövegdoboz, amibe az optimalizálandó GLSL kódot kell másolni, ki kell választani a GLSL verziót, majd kattintani kell az *Optimize!* feliratú gombra. Az alkotói szerint ez a következő (GPU-független) optimalizációkat hajtja végre[32]:

- Függvények behelyettesítése (*function inlining*, lásd: 3.4.3)
- Nem használt kódrészek eltávolítása (*dead code removal*)
- Konstans kifejezések fordítási időben való kiszámítása (*constant folding*)
- *Copy propagation*
- *Constant propagation*
- Aritmetikai optimalizáció (*arithmetic optimization*)
- Stb.

A tesztek alapján az így kapott kód a futási teljesítményen csak minimálisan (5-10%-kal) javított, néhány (jellemzően gyengébb) hardveren ugyanennyit rontott. Ami fontosabb, hogy a fordítási idő azonban rengeteget javult, ezért a webes verziók az így kapott kóddal jóval hamarabb betöltődnek (csak a WebGL 2.0-ás verzióval, lásd: 2.1.2, az 1.0-ával nem működik megfelelően az optimalizáló). Ami még fontosabb, hogy ez valami különös oknál fogva megoldotta a heisenbugot (lásd: 3.9): az optimalizált kóddal már minden általam tesztelt számítógépen (ahol működnie kell) hibátlanul működik a program.

Így tehát (a WebGL 1.0-ás verziók kivételével) mindenhol a generátor által készített kódok futnak, de természetesen meghagytam mellettük az eredeti, „nyers” forrást is, hiszen az jóval érhetőbb, programozói szemmel olvashatóbb kód, valamint a függvénybehelyettesítésekkel, cikluskibontásokkal járó többlet miatt jóval rövidebb is.

Az, hogy a heisenbugot ez miért oldotta meg, számomra egyelőre továbbra is rejtély marad. Sejtésem szerint az érintett Nvidia kártyákra írt fordító (Windows alatt) valami olyan dolgot csinál másképp, amit minden más platformon minden fordító egymáshoz képest azonosan hajt végre. Próbáltam ezért *nem definiált viselkedés*hez (angolul: *undefined behaviour*), és *nem specifikált viselkedés*hez (angolul: *unspecified behaviour*) vezető kifejezéseket keresni az eredeti kódban, de e sorok írásakor ez még nem vezetett eredményre.

3.10.1. További optimalizációs lehetőségek

Az egyik legnépszerűbb optimalizációs lehetőség a sugárkövetéshez az ún. *befoglaló keretek* (angolul: *bounding volume*) alkalmazása. Ez a jelenlegi munkámban nincs benne, de a jövőben majd mindenképpen szeretném beleépíteni. Ennek lényege a következő lépésekkel magyarázható el:

- Sok kisebb (vagy akár egy nagy és bonyolult) objektumot határoljunk körbe egyetlen, egyszerű (pl.: gömb, vagy téglatest) objektummal.
- Először csak a határoló objektummal keressünk metszéspontot. Ha ezzel nem találunk, abba is hagyhatjuk a keresést, mivel a többi objektum ezen belül van, ezért azokkal sem találhatunk metszést. Ezzel sok felesleges számolástól megkímélhetjük a programot, jelentősen felgyorsítva annak futását.
- Ha találtunk metszést, akkor kénytelenek vagyunk a többi (belső) objektumra is meghívni a metszéspontkeresést.

Ezt aztán kiterjeszthetjük az egész színtérre, *hierarchikus befoglaló kereteket* hozhatunk létre, melyek fa struktúrát alkotnak. Ezekben egy részfát csak akkor kell kiértékelni, ha a gyökérrel van metszés. Ennek leghatékonyabb implementációit jellemzően *oktális fa* (angolul: *octree*), vagy még inkább *k-d fa* (angolul: *k-d tree*) alkalmazásával szokták megvalósítani.

3.11. Tesztelés

A beviteli adatok sokfélesége egy ilyen jellegű grafikus programnál igencsak korlátozott, ugyanakkor az emberi szem jóformán nélkülözhetetlen a tesztelésükhöz. A program futását igyekeztem a lehető legtöbb féle számítógépen letesztelni (a natív, és a webes verziókat egyaránt). Többször találkoztam olyannal, hogy míg az egyik gépen jól futott a program, a másikon le sem fordultak a shaderek, vagy futáskor csak egyetlen, rózsaszínes színt jelenítettek meg. Ezek a számítógépekben működő különböző GPU-któl (és illesztőprogramjaiktól), valamint a GLSL különböző verzióitól függő jelenségek voltak. Hogy egy konkrét példát is említsek: bizonyos AMD GPU-k esetében a shaderek nem fordultak le, ha a vertex attribútumok előtt az *in* kulcsszó szerepelt, ki kellett cserélni őket az *attribute* kulcsszóra. Hasonlóan a *varying*, és az *in*/*out* kulcsszavakkal: némelyik GLSL verzió nem engedte az egyiket, némelyik a másikat. Sok változat kipróbálása, tesztelése kellett ahhoz, hogy végül a natív és a webes verziók is megfelelően működjenek az Intel, az AMD, és az Nvidia GPU-kkal is.

A webes változatok különböző böngészőkkel való tesztelése viszonylag sok időt vett igénybe. Érdekesség, hogy Windows alatt a böngészők jellemzően sokkal több ideig töltik a programot, mint Linux alatt (ez talán a DirectX-szel való együttműködés megvalósítása miatt lehet Windows alatt). Ez a hosszú töltés a Chromium alapú böngészők esetében a WebGL kontextus elvesztését okozza 20 másodperc után. Ezen aztán jelentősen javított az az automatikus optimalizáció (lásd: 3.10), ami viszont csak a WebGL 2.0-ás verzióin működik; így ismét hangsúlyozom, hogy erősen ajánlott ennek a lehetőségnek az aktiválása (lásd: 2.1.2). A futási teljesítményen nem vettem észre jelentős különbséget a WebGL 1.0 és a 2.0 között, ugyanakkor a töltési idő egyértelműen sokkal rövidebb a 2.0-át használva. A WebGL 2.0 remélhetőleg hamarosan minden böngészőben alapértelmezettként be lesz kapcsolva.

A tórusz megjelenésével eleinte voltak problémák, erről egy külön fejezetet is írtam (lásd: 3.7). Ennek megoldása is sok tesztelést igényelt a különböző platformok mindegyikén.

A Windowsos, illetve Linuxos installallereket igyekeztem a lehető legtöbb, számomra elérhető számítógépen kipróbálni, frissen telepített rendszereken tesztelni.

A térben a mozgás teljesen szabad, kameránkkal belemehetünk bármilyen testbe, nincsen semmiféle ütközés (angolul: *collision*) megvalósítva. Akár kimehetünk a skybox-on kívülre is (ehhez a távolságot figyelembe véve több, mint 10 percig kell haladnunk a sík felé). Ilyen esetekben a kép torzulhat, a skybox síkjai láthatóvá, egymástól elkülöníthetővé válnak. A szabad mozgást azonban ezen jelenségek miatt nem kívánom korlátozni.

3.11.1.Benchmarking

Az aktuális FPS számot minden verziónál kiírja a program. Elsősorban ezekkel jellemezhető, hogy a program futása mennyire folyamatos az adott rendszeren. Az FPS adatokat a következő táblázatokba gyűjtöttem össze, a velem egy háztartásban lévő eszközöket felhasználva. Mivel a futási teljesítmény elsősorban a GPU-tól függ, ezért azok típusával jellemeztem az eszközöket:

Natív (OpenGL):

LITE version Depth: 8 Shadows: OFF				
	Windows (800x600)	Ubuntu (800x600)	Windows (1920x1080)	Ubuntu (1920x1080)
Intel HD 4400	45 FPS	71 FPS	11 FPS	18 FPS
Nvidia GT 840M	142 FPS	143 FPS	46 FPS	43 FPS
Nvidia GTX 560 Ti	360 FPS	352 FPS	105 FPS	102 FPS
Nvidia GTX 1070	1093 FPS	1197 FPS	332 FPS	391 FPS

LITE version Depth: 8 Shadows: ON				
	Windows (800x600)	Ubuntu (800x600)	Windows (1920x1080)	Ubuntu (1920x1080)
Intel HD 4400	29 FPS	39 FPS	7 FPS	10 FPS
Nvidia GT 840M	73 FPS	72 FPS	21 FPS	20 FPS
Nvidia GTX 560 Ti	132 FPS	146 FPS	38 FPS	42 FPS
Nvidia GTX 1070	416 FPS	414 FPS	123 FPS	116 FPS

PRO version Depth: 8 Shadows: OFF				
	Windows (800x600)	Ubuntu (800x600)	Windows (1920x1080)	Ubuntu (1920x1080)
Intel HD 4400	14 FPS	8 FPS	4 FPS	2 FPS
Nvidia GT 840M	28 FPS	32 FPS	10 FPS	11 FPS
Nvidia GTX 560 Ti	43 FPS	46 FPS	14 FPS	17 FPS
Nvidia GTX 1070	213 FPS	216 FPS	78 FPS	75 FPS

PRO version Depth: 8 Shadows: ON				
	Windows (800x600)	Ubuntu (800x600)	Windows (1920x1080)	Ubuntu (1920x1080)
Intel HD 4400	5 FPS	2 FPS	1 FPS	<1 FPS
Nvidia GT 840M	9 FPS	10 FPS	3 FPS	3 FPS
Nvidia GTX 560 Ti	14 FPS	14 FPS	5 FPS	5 FPS
Nvidia GTX 1070	77 FPS	71 FPS	23 FPS	22 FPS

Web (WebGL 1.0):

LITE version Depth: 8 Shadows: OFF				
	Windows (800x600)	Ubuntu (800x600)	Windows (1920x1080)	Ubuntu (1920x1080)
Intel HD 4400	46 FPS	43 FPS	12 FPS	14 FPS
Nvidia GT 840M	60+ FPS	55 FPS	40 FPS	16 FPS
Nvidia GTX 560 Ti	60+ FPS	60+ FPS	60+ FPS	30 FPS
Nvidia GTX 1070	60+ FPS	60+ FPS	60+ FPS	60+ FPS

LITE version Depth: 8 Shadows: ON				
	Windows (800x600)	Ubuntu (800x600)	Windows (1920x1080)	Ubuntu (1920x1080)
Intel HD 4400	28 FPS	28 FPS	7 FPS	9 FPS
Nvidia GT 840M	59 FPS	34 FPS	22 FPS	9 FPS
Nvidia GTX 560 Ti	60+ FPS	30 FPS	52 FPS	15 FPS
Nvidia GTX 1070	60+ FPS	60+ FPS	60+ FPS	60+ FPS

PRO version Depth: 8 Shadows: OFF				
	Windows (800x600)	Ubuntu (800x600)	Windows (1920x1080)	Ubuntu (1920x1080)
Intel HD 4400	2 FPS	6 FPS	<1 FPS	1 FPS
Nvidia GT 840M	13 FPS	22 FPS	4 FPS	8 FPS
Nvidia GTX 560 Ti	32 FPS	30 FPS	12 FPS	10 FPS
Nvidia GTX 1070	60+ FPS	60+ FPS	28 FPS	32 FPS

PRO version Depth: 8 Shadows: ON				
	Windows (800x600)	Ubuntu (800x600)	Windows (1920x1080)	Ubuntu (1920x1080)
Intel HD 4400	1 FPS	2 FPS	<1 FPS	<1 FPS
Nvidia GT 840M	4 FPS	10 FPS	1 FPS	2 FPS
Nvidia GTX 560 Ti	12 FPS	12 FPS	28 FPS	3 FPS
Nvidia GTX 1070	36 FPS	50 FPS	10 FPS	12 FPS

Web (WebGL 2.0):

LITE version Depth: 8 Shadows: OFF				
	Windows (800x600)	Ubuntu (800x600)	Windows (1920x1080)	Ubuntu (1920x1080)
Intel HD 4400	33 FPS	48 FPS	8 FPS	11 FPS
Nvidia GT 840M	60+ FPS	60+ FPS	46 FPS	32 FPS
Nvidia GTX 560 Ti	60+ FPS	60+ FPS	60+ FPS	60+ FPS
Nvidia GTX 1070	60+ FPS	60+ FPS	60+ FPS	60+ FPS

LITE version Depth: 8 Shadows: ON				
	Windows (800x600)	Ubuntu (800x600)	Windows (1920x1080)	Ubuntu (1920x1080)
Intel HD 4400	24 FPS	29 FPS	6 FPS	7 FPS
Nvidia GT 840M	60+ FPS	60+ FPS	30 FPS	18 FPS
Nvidia GTX 560 Ti	60+ FPS	60+ FPS	52 FPS	30 FPS
Nvidia GTX 1070	60+ FPS	60+ FPS	60+ FPS	60+ FPS

PRO version Depth: 8 Shadows: OFF				
	Windows (800x600)	Ubuntu (800x600)	Windows (1920x1080)	Ubuntu (1920x1080)
Intel HD 4400	9 FPS	5 FPS	2 FPS	2 FPS
Nvidia GT 840M	18 FPS	25 FPS	5 FPS	8 FPS
Nvidia GTX 560 Ti	32 FPS	30 FPS	10 FPS	12 FPS
Nvidia GTX 1070	60+	60+	46 FPS	52 FPS

PRO version Depth: 8 Shadows: ON				
	Windows (800x600)	Ubuntu (800x600)	Windows (1920x1080)	Ubuntu (1920x1080)
Intel HD 4400	2 FPS	2 FPS	<1 FPS	<1 FPS
Nvidia GT 840M	6 FPS	8 FPS	1 FPS	2 FPS
Nvidia GTX 560 Ti	10 FPS	10 FPS	3 FPS	3 FPS
Nvidia GTX 1070	46 FPS	60 FPS	18 FPS	15 FPS

Ezek alapján nem igazán lehet győztest hirdetni az örökös Windows-Linux platformháborúban; hol az egyiken mértem több FPS-t, hol a másikon. Ami meglepő lehet, hogy a natív verziók előnye a webes verziókhoz képest sem minden esetben egyértelmű, sőt bizonyos kártyákkal, bizonyos esetekben határozottan jobb eredményt mértem a weben, habár nem ez volt a jellemző.

További érdekesség, hogy a webes verziókat tableten (Nvidia Shield Tablet), és okostelefonon (OnePlus 3) is sikerült betöltenem, ráadásul egészen élvezhető teljesítményt produkáltak, ám irányítani (gombok híján) nem tudtam egyiken se.

4. Konklúzió

Vitathatatlan, hogy a GPU-k jelentősége egyre nagyobb a számítógépes világban. A sokadik nekifutásra, nemrégiben újra megjelent VR sisakokkal a nagy erejű GPU-kra való igény csak tovább növekszik, sőt mapanság már egy okostelefontól is elvárjuk, hogy szaggatás nélkül jelenítsen meg nagy felbontású filmeket, lenyűgöző grafikájú játékokat.

Ráadásul nem csak az egyre nagyobb felbontású, egyre nagyobb frissítési rátájú kijelzők, vagy az egyre szebb grafikájú programok, játékok igénylik az erős GPU-kat. Újabban ezek az eszközök olyan tudományos célokra is felhasználhatók, mint például a mesterséges intelligenciában való kutatás; ebbe beleértve a valós idejű beszédfelismerést, önjáró autók navigációját, stb. Bizonyos *deep learning*, és *machine learning* technológiákkal működő programok futásának sebességét nagymértékben képesek növelni a megfelelő GPU-k; 10, 20, vagy akár több, mint 30-szoros gyorsulást is elérhetnek a CPU-khoz képest.[37]

Mindeközben a natív alkalmazások, és a webes alkalmazások közti „szakadék" már szinte csak egy „gödör", amely sokkal könnyedébben áthidalható, mint ezelőtt 5-10 évvel; és ez a különbség várhatóan a WebAssembly megjelenésével csak tovább fog csökkenni (lásd: 3.1.2). Jelenleg azonban még a tapasztalat azt mutatja, hogy a teljesítményorientáció megköveteli a natív programok írását.

Ha azonban hajlandóak vagyunk egy kis kompromisszumot kötni, akkor a programunkat élvezhetjük egy webböngészőben, ennek minden előnyével együtt (a teljesség igénye nélkül):

- A programunkat nem szükséges lefordítani a különböző platformokra, bárhol működni fog, ahol modern webböngésző működik, és internetelérés van.
- Így a programunkat nem szükséges (a szó köztudatban lévő értelmében) sem letölteni, sem telepíteni (WebGL esetén még plugin telepítés sem kell).
- Így megkíméljük magunkat rengeteg plusz munkától is.

5. Irodalomjegyzék

[1] Szirmay-Kalos László, Antal György, Csonka Ferenc: Háromdimenziós grafika, animáció és játékfejlesztés, ComputerBooks, 2009, [486], ISBN-963-618-303-1

[2] OpenGL | Wikipedia
https://hu.wikipedia.org/wiki/OpenGL
2016-10-09

[3] OpenGL | Wikipedia
https://en.wikipedia.org/wiki/OpenGL
2016-10-09

[4] Vulkan (API) | Wikipedia
https://en.wikipedia.org/wiki/Vulkan_(API)
2016-10-11

[5] Számítógépes grafika házi feladat tutorial | BME VIK
http://vik.wiki/Számítógépes_grafika_házi_feladat_tutorial
2016-11-09

[6] WebGL | Wikipedia
https://en.wikipedia.org/wiki/WebGL
2016-10-11

[7] WebGL | Wikipedia
https://hu.wikipedia.org/wiki/WebGL
2016-10-11

[8] WebAssembly | Wikipedia
https://hu.wikipedia.org/wiki/WebAssembly
2016-10-11

[9] Rasztergrafika | Wikipedia
https://hu.wikipedia.org/wiki/Rasztergrafika
2016-10-12

[10] RGB színtér | Wikipedia
https://hu.wikipedia.org/wiki/RGB_színtér
2016-10-12

[11] Ray tracing alapok | Kapronkai Tamás
http://rs1.szif.hu/~tomcat/konf/rtalap/rtalapok.htm
2016-10-12

[12] Screen tearing | Wikipedia
https://en.wikipedia.org/wiki/Screen_tearing
2016-10-14

[13] Ray tracing primitives | University of Cambridge
https://www.cl.cam.ac.uk/teaching/1999/AGraphHCI/SMAG/node2.html
2016-10-17

[14] Möller-Trumbore intersection algorithm | Wikipedia
https://en.wikipedia.org/wiki/Möller-Trumbore_intersection_algorithm
2016-10-17

[15] Torus intersection | Íñigo Quílez
https://www.shadertoy.com/view/4sBGDy
2016-10-17

[16] UV mapping | Wikipedia
https://en.wikipedia.org/wiki/UV_mapping
2016-11-02

[17] Bump mapping | Wikipedia
https://en.wikipedia.org/wiki/Bump_mapping
2016-11-02

[18] GLSL rotation about an arbitrary axis | Neil Mendoza
http://www.neilmendoza.com/glsl-rotation-about-an-arbitrary-axis
2016-11-02

[19] The Celestia Motherlode: Earth Surface Maps
http://www.celestiamotherlode.net/catalog/earth.php
2016-11-02

[20] Textures | Massachusetts Institute of Technology
http://imbrium.mit.edu/EXTRAS/CELESTIA/global
2016-11-02

[21] Gravity WebGL | Vichnál Valentin
https://github.com/valentinvichnal/gravity.js
2016-11-02

[22] NormalMap Online | Cpetry
http://cpetry.github.io/NormalMap-Online
2016-11-02

[23] Fényvisszaverődés | Wikipedia
https://hu.wikipedia.org/wiki/Fényvisszaverődés
2016-11-09

[24] Fénytan | Fizkapu
http://www.fizkapu.hu/fizfoto/fizfoto6.html
2016-11-09

[25] Fénytörés | Wikipedia
https://hu.wikipedia.org/wiki/Fénytörés
2016-11-09

[26] Snell's Law Poster | fineartamerica
http://fineartamerica.com/pSroducts/refraction-and-total-internal-reflection-giphotostock-poster.html
2016-11-09

[27] Manual | OpenGL 4.x
https://www.opengl.org/sdk/docs/man4/html
2016-11-09

[28] Diplomarbeit | Günther Voglsam
https://www.cg.tuwien.ac.at/research/publications/2013/Voglsam_2013_RRT/Voglsam_2013_RRT-Thesis.pdf
2016-11-09

[29] Sugárkövetés előadás fóliák | BME VIK (Szirmay-Kalos László)
http://cg.iit.bme.hu/portal/oktatott-targyak/szamitogepes-grafika-es-kepfeldolgozas/sugarkoevetes
2016-11-09

[30] Heisenbug | Wikipedia
https://en.wikipedia.org/wiki/Heisenbug
2016-11-17

[31] GLSL Optimizer | Joshua Koo
https://zz85.github.io/glsl-optimizer
2016-11-17

[32] GLSL Optimizer | Joshua Koo (github page)
https://github.com/zz85/glsl-optimizer
2016-11-17

[33] WebGL Tutorial Videos | Kamaron Peterson
https://www.youtube.com/user/IntroTutorials1/videos
2016-11-20

[34] glMatrix | Brandon Jones
http://glmatrix.net
2016-11-20

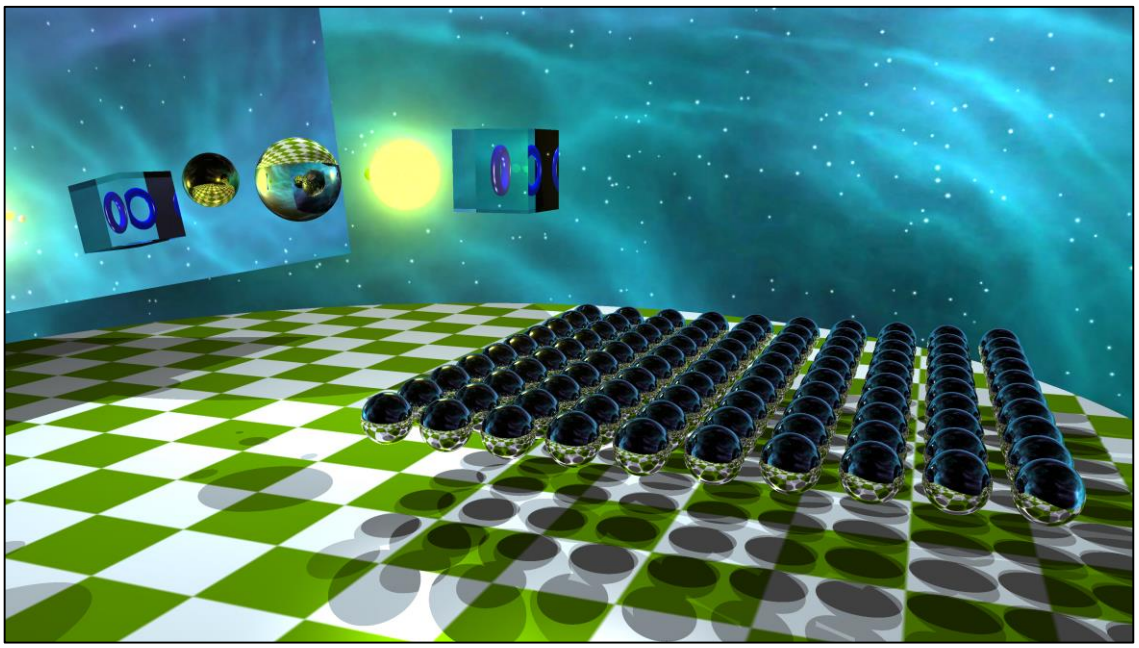
[35] Using requestAnimationFrame | Chris Coyier
https://css-tricks.com/using-requestanimationframe
2016-11-20

[36] GeForce GTX 1080 vs GeForce 9800 GT | HWBench
http://hwbench.com/vgas/geforce-gtx-1080-vs-geforce-9800-gt
2016-11-20

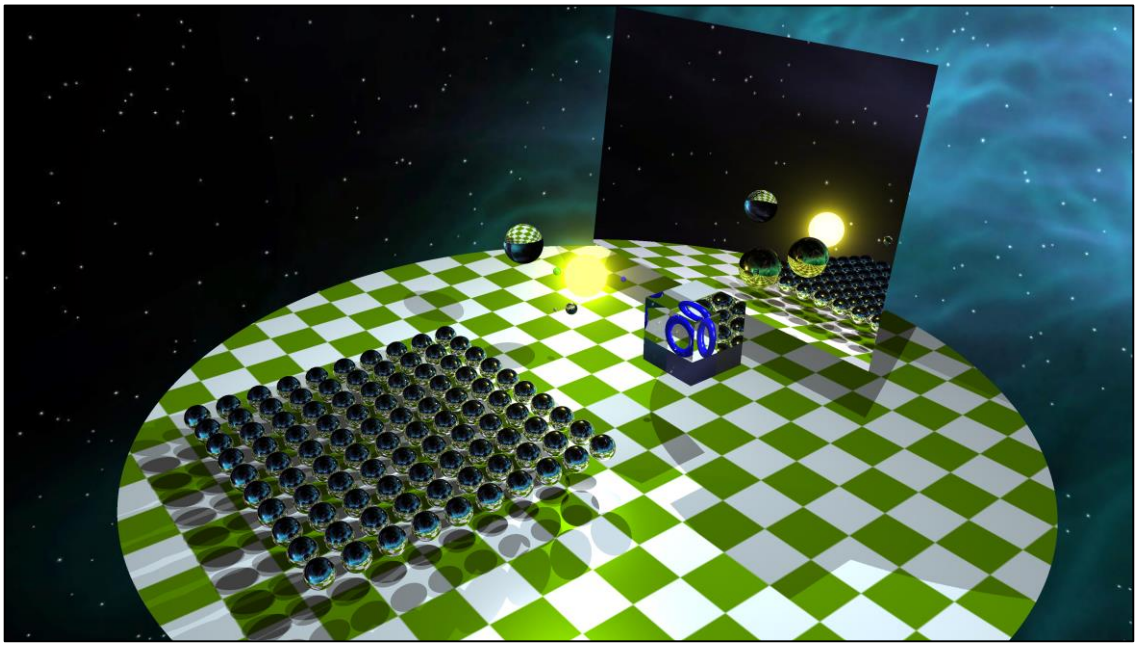
[37] Machine Learning – GPU Accelerated Applications | NVIDIA
http://www.nvidia.com/object/machine-learning.html
2016-11-21

6. Mellékletek

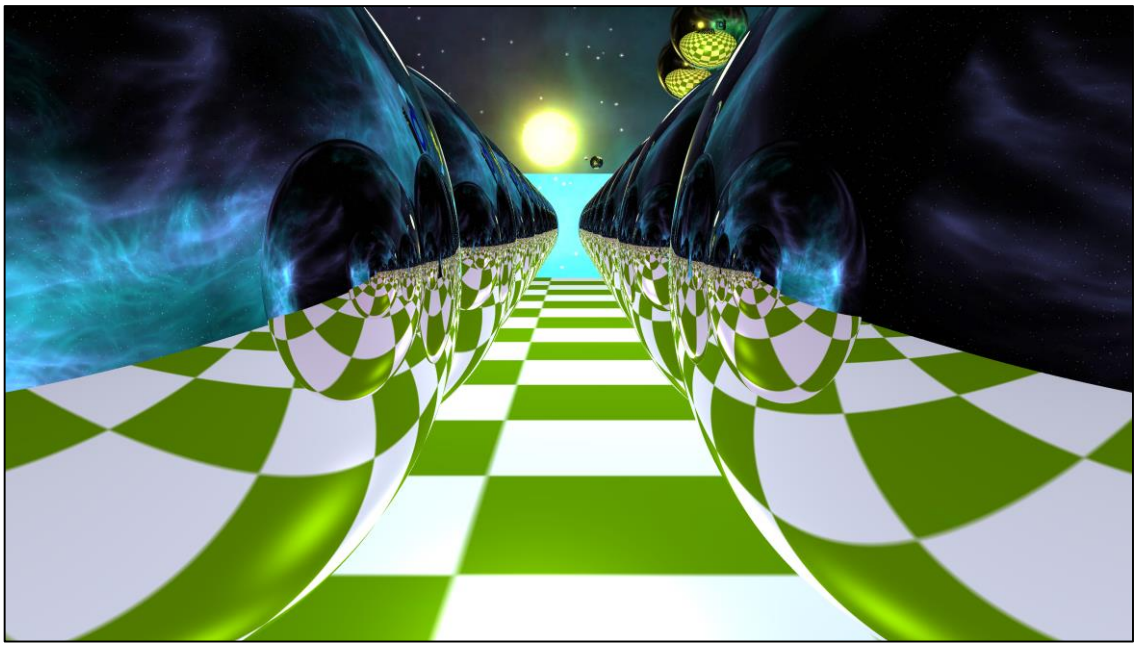
Végezetül bemutatok néhány képet a programommal kapcsolatosan:



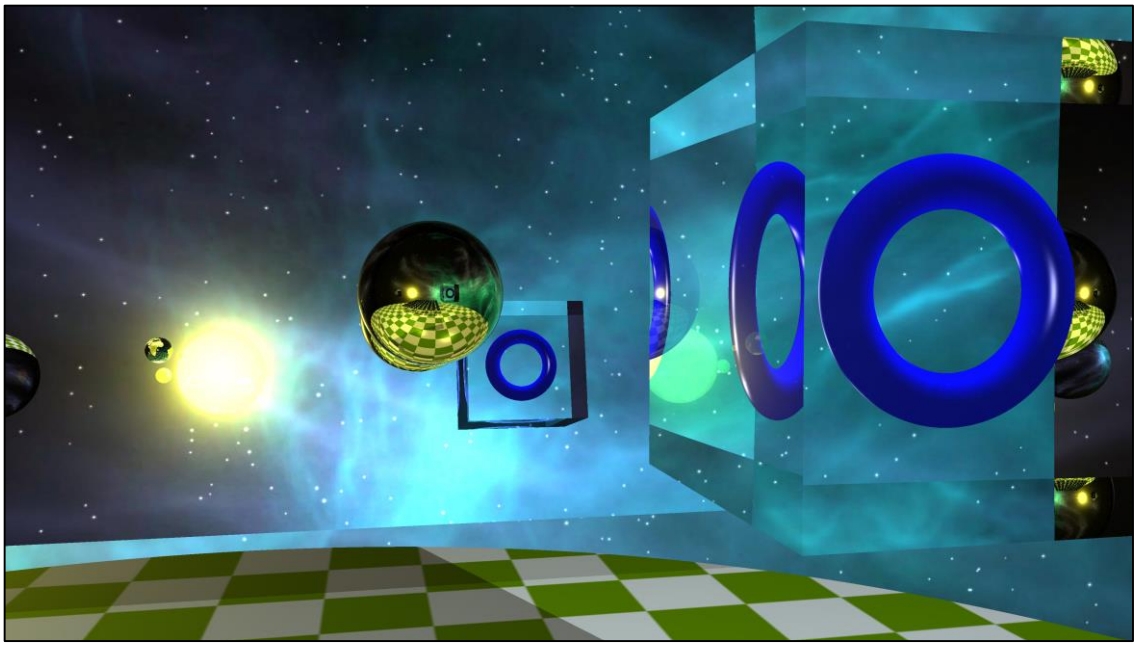
43. ábra. *Valós idejű sugárkövetés több, mint 100 gömböt használva (pro verzió)*



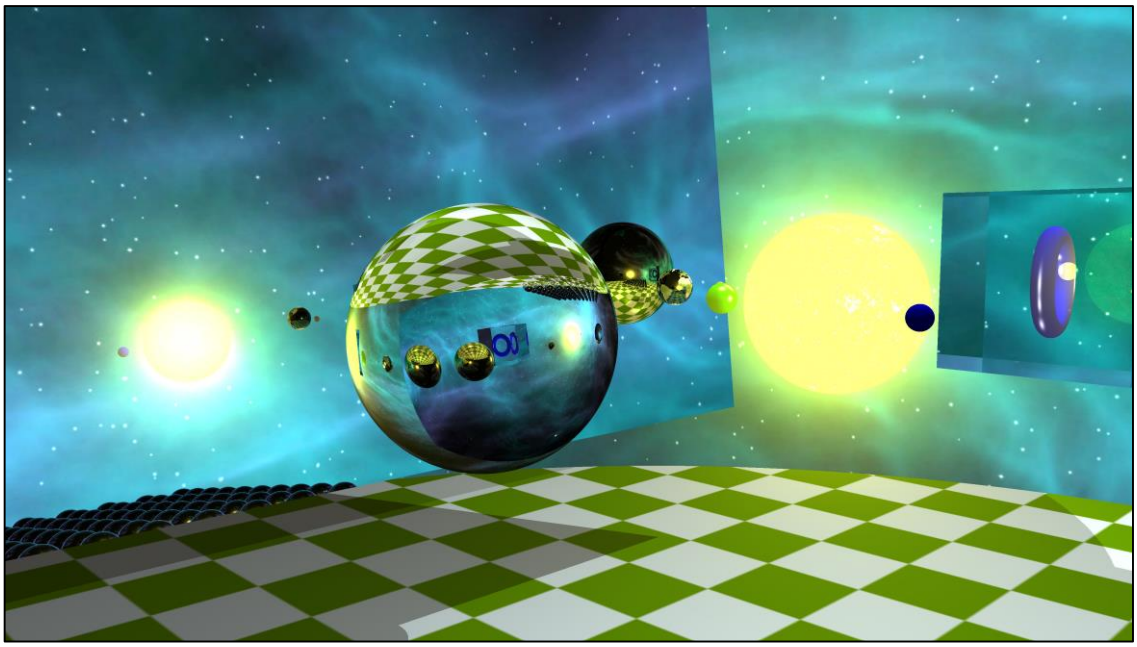
44. ábra. *Valós idejű sugárkövetés másik nézetből (pro verzió)*



45. ábra. *„Infinity mirror”*



46. ábra. *Az üvegkockán jól láthatóak mind a tükör-, mind a törésirányú fények*



47. ábra. *Az üveggömbön úgy törik a fény, hogy a kapott kép „fejjel lefelé” jelenik meg*

48. ábra. *Üveggömb a valóságban, a Parlamenttel (Budapest)*