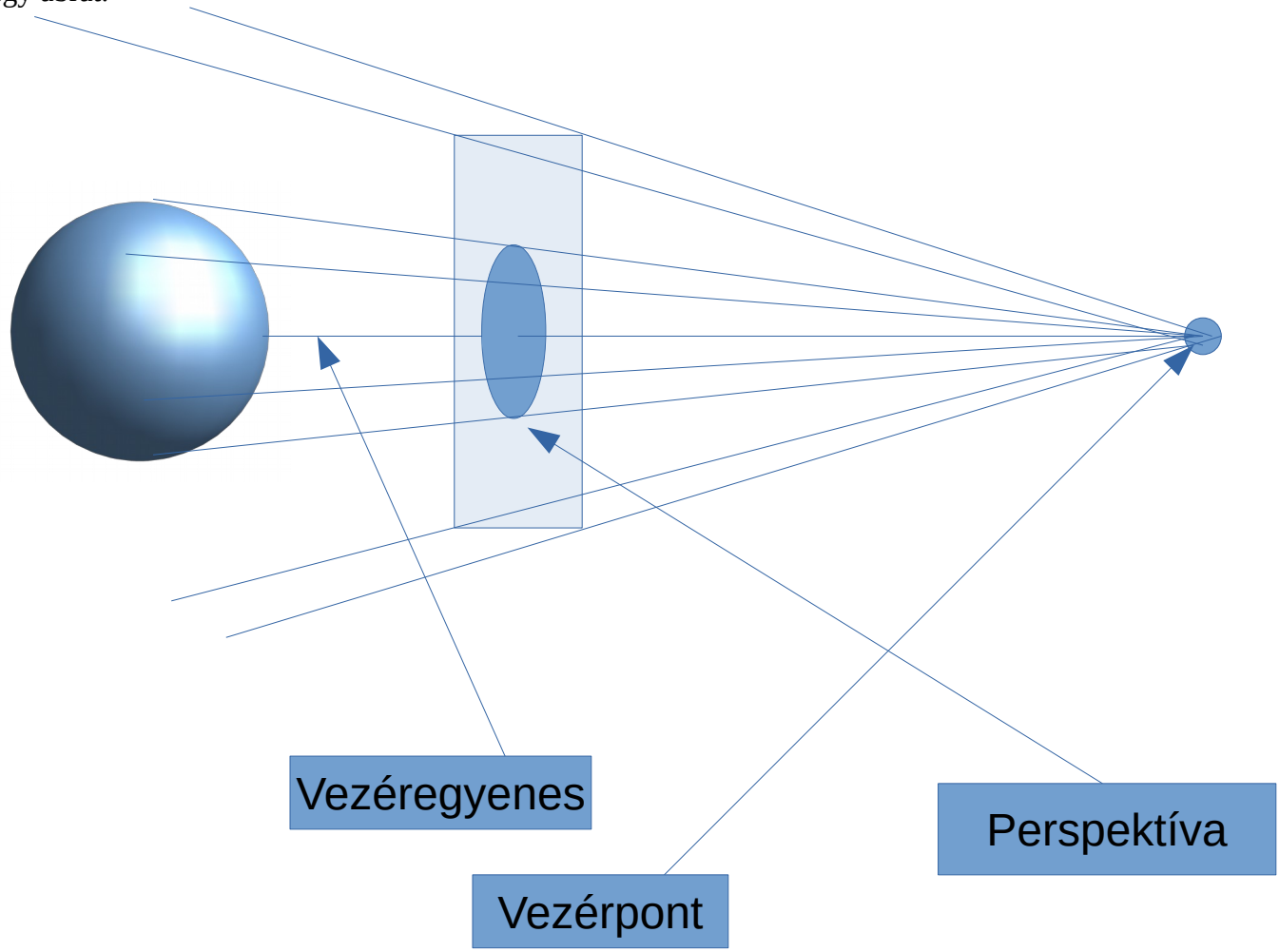
Ray Tracing fejlesztői dokumentáció

Lázár György, Neptun: LIJIA8

1. Az alapvető matematikai elképzelés

A program teljes egészében matematikai alapokon fekszik. Első körben létrehozunk egy teret elméletben, amibe helyezzük az objektumainkat. Minden egyes objektumnak matematikai egyenletekkel leírhatónak kell lennie. Vegyük például a gömböt, aminek egyenlete $(x-x_o)^2+(y-y_o)^2+(z-z_o)^2 = r^2$. Innen már tudunk számításokat elvégezni, például ki tudjuk számolni egy gömb-egyenes metszéspontját. Az egyenes egyenlete $(x-x_o)/A = (y-y_o)/B = (z-z_o)/C$. Ha sok ilyen egyenest alkotnánk, akkor ezek közül néhány akár metszheti is a gömböt. Ha a gömb felé irányítjuk ezeket az egyeneseket, akkor ha kiválasztjuk ezek közül azokat, amik elmetszik a gömböt, akkor ha ezek közé még beteszünk egy síkot, az egy kört fog kirajzolni. Mivel ezt így elég nehéz elmagyarázni, ezért készítettem hozzá egy ábrát.



A gyakorlatban ehhez rendezett és véges számú egyenest kell létrehoznunk. Ezért a perspektíva síkjának méretét meghatározzuk *height* és *width* változók értékei szerint (ezeket még megszorozva *perspPixelSize* értékével fogjuk kapni aktuálisan a perspektíva méretét, egységekben kifejezve), majd felosztjuk a perspektívát négyzethálósan, méghozzá úgy, hogy ez a négyzetháló height*width számú metszéspontot tartalmazzon, és minden oldala perspPixelSize legyen. Így már tudunk sugarakat húzni a

vezérpontból a perspektíva pontjaihoz. Ezt a folyamatot a calcPersp(struct ray r) függvény végzi, ami paraméterként kapja a vezérsugarat. A pontos vektorműveletes számítási folyamat a forráskódban dokumentálva megtalálható. Végül a függvény eltárolja a perspektívát úgy, hogy minden sugarat egy 2 dimenziós ray tömbbe teszi, ahol (a felhasználó nézőpontjából) a [0][0] sugár a bal felső pixelnek felel meg, majd [height][width] formátumban a további sugarak is megfeleltethetőek egy-egy pixelnek a képernyőn. A sugarak ütköztetését “magas szinten” az engage(struct ray r) függvény végzi, ami végignézi egy sugár összes ütközési pontját az összes objektummal, és kiválasztja a legközelebbit a vezérponthoz, majd ezt a megfelelő struktúrával visszaadja. Erről részletesebben később tárgyalok.

2. A program (nagyon) egyszerűsített felépítése

A program, mint minden valamire való C program a main(/*argumentumok*/) függvény hívásával indul. A main függvény inicializálja a programot, és elindítja az első szálat, a subthreadet, aminek ezután lényegében átadja az irányítást, ugyanis a main ezentúl már csak a felhasználói bemenet figyelését fogja végezni. A subthread kezeli a perspektívával kapcsolatos számításokat (a main szál által megadott adatok alapján forgatja, eltolja, stb), illetve a ray tracing szálakat is ez kezeli (továbbiakban csak RT). Az RT szálak által kiszámolt pontokból (pixels[][] tömb) is a subthread rajzolja ki a képet a képernyőre. A térhatású megjelenést úgy érem el, hogy a sugarak beesési szögének függvényében egy adott pontot sötétebbé vagy világosabbá teszek. A fényforrást úgy veszem figyelembe, hogy az ütközési pontba húzok még egy sugarat, elindítva a fényforrásból. Ennek szöge alapján számolom ki matematikailag nem pontosan a fényerősséget. A kettő sugár által alkotott fényerősség-értéket szintén egy matematikailag nagyon nem pontos, de egyszerű, és látványos hatást elérő képlettel átlagolom, így létrehozva az adott pont tényleges színét.

3. A szálak értelme, szükségessége

A szálak szerves részét képezik a programnak, ha bár hasonlót lehetett volna írni egyetlen szálon futtatva is. Az első, és nyílvánvaló ok, az az, hogy a felhasználói bemenet figyelésének és a kép kirajzolásának nagyjából egyidejűleg kell történnie. Ez a probléma már önmagában szálak használatáért kiált. A másik érvem a szálak mellett, hogy a mai standardok mellett egy viszonylag erős és sok processzor maggal rendelkező laptoppal végeztem a tesztelés és programozás nagy részét (6 fizikai mag, 12 virtuális szál), így a teljesítmény, amit a párhuzamosításokból nyertem, nagyon nem elhanyagolható. Az RT szálak számát beállíthatjuk a programban az RTThreads opcióval a konfigurációs, vagy a map fájlban is akár. Egy egy gömböt és egy síkot tartalmazó map esetén a laptopom (töltőn, 70 W CPU power limit-tel, alap beállításokkal (linuxon nehéz undervoltolni, de egy --145 mV offset feszültséggel is kipróbálnám szívesen), rövid távon, 750x500-as felbontással) 11 RT szállal 22-24 FPS-t volt képes a szemünk elé prezentálni, míg egyetlen eggyel 7-8-at. Innen látszik, hogy az életünket mennyire képes megkönnyíteni a több szálon való futtatás (a teszt kicsit még így is csal, hiszen a perspektíva kiszámítása mindenképpen külön szálon történik, így 1 RT szállal is gyakorlatilag a számításokat 3 szálon végezzük). Mivel a Kármán Tódor Kollégiumban tagja vagyok a számítástechnikai szakosztálynak (próbásként), ezért van jogosultságom egy 24 maggal, és 48 szállal rendelkező szerver gép elérésehez. Ezen a programom egy lecsupaszított változatát futtattam (titokban) (az SDL könyvtár teljes kiiktatásával, így csak a számításokat végezte el a program, és kiírt egy fps értéket, hiszen a szerver gépen grafikus felület nem fut), ami ugyan ezekkel a beállításokkal, csak 24 szálon képes volt 40 körüli képkockasebességet produkálni. Ezért ezekkel az érvekkel szeretném megmagyarázni a szabványos és hordozható kódtól való eltérést, és a külső könyvtárak használatát.

4. A szálak részletesebb leírása

Kezdjük a main szállal. Ez a szál az legelején beolvassa a megadott paramétereket, és ezt egy globális változóban eltárolja (igen, tudom, ezt el lehetne kerülni azzal, ha az init()-nek megadnék paramétert, de úgy gondolom, hogy a perspektíva és a pixelek eltárolása mellett (amik ugyebár már 750x500-as felbontás mellett is 375000 tagból állnak egyenként, és tartalmaznak egy halom változót, köztük doubleöket is) ez a karaktertömböcske elhanyagolható). Ez után az init() segítségével inicializálja a programot, beállítja néhány változó értékét, lenulláz és létrehoz néhány tömböt. Ellenőrzi továbbá, hogy megadtunk-e fájlt megnyitásra. Majd elindítja a subthreadet, és ezután elkezdi figyelni a felhasználói bemenetet SDL eventekkel. Ha SDL_QUIT eventet kap, akkor beállítja a sigterm boolean változó értékét true-ra, és elkezdi bevárni a subthreadet. A subthread is elkezd ekkor a sigterm hatására leállni, ő is bevárja az általa elindított szálakat. Miután a subthread leállt, felszabadítjuk a dinamikusan lefoglalt memóriát, majd kilépünk az SDL-ből, és leállítjuk a programot. Az SDL_QUIT event például akkor jelentkezik, ha a felhasználó az X gombra kattint az ablakban. A szálak bevárására a kilépés előtt azért van szükség, mert enélkül a program random segmentation faultokat dobált kilépésnél.

Folytassuk a subthread-del. Már említettem, hogy a nevével ellentétben az RT utáni legfontosabb feladatokat ez a szál látja el. Ez indítja, paraméterezi, és általánosan kezeli az RT és perspektíva számoló szálakat, ez végzi a perspektíva vezérsugarának irányának és pozíciójának kiszámítását, valamint az egyéb kiírásokat a képernyőre (fps, pozíció). Ez hajtja végre a reset-et is (R betű, nincs benne a felhasználói kézikönyvben, mert nem fontos, és mert nem fért bele). Amikor a sigterm true lesz, bevárjuk a perspektíva számoló szálat, felszabadítjuk az RT szálaknak fenntartott dinamikus memóriát (RT szál ilyenkor nem futhat), és kilépünk a szálból.

Végezetül pedig az RT szál(ak). A perspektíva kiszámolását végző szálról a függvények részben tárgyalok inkább, oda jobban passzol, mivel nem csinál semmit, csupán meghívja azt a függvényt, amelyik kiszámolja a perspektívát. Mintha a program lineárisan futna, csak így gyorsabb lesz. Visszatérve az RT szálakra: Ezek adják a program szívét. Itt dől el, hogy mikor ütközik egy sugár, mivel ütközik a sugár, stb. Az RT szálak nagyon nagymértékben függenek az engage(struct ray r) függvénytől, amit szintén fogok részletesebben is tárgyalni. Egyelőre erről annyit, hogy ez a függvény egy adott sugarat félegyenesként kezelve, a megadott pontját véve kiindulópontnak megadja, hogy melyik pontban ütközik (és hogy mivel, de ez csak ott fontos, hogy az árnyékok ne látszanak át a síkokon, más helyen nem számít hogy az adott sugár mivel ütközik) legelőször a sugár. Ha nem ütközik semmivel se, akkor a rayColl structnak (amit visszaad a függvény) a h boolean eleme false lesz. Amennyiben ez nem false, egy halom egyéb ellenőrzést, valamint a fényforrásból származó sugár kiszámítása után számolunk az adott pontba egy fényerősséget, majd azt a pixels[][] tömbbe eltároljuk. A párhuzamosított szálak úgy működnek, hogy az elsőként meghívott szál számolja a sorban az első elemet, majd ezután annyi elemet ugrik, ahány szál van összesen. Így 11 szál esetén az első szál számolja az első, a 12., a 33., stb pixeleket. A sor végén lejjebb ugrik egy sort, úgy, hogy ez a folytonosság ne szakadjon meg. Így minden szál nagyjából egyforma mennyiségű elemet fog számolni, ezzel gyorsítva meg a program lefutását. A subthreadben bevárjuk ezeket a szálakat, addig nem kezdhetjük el a perpsektíva újbóli kiszámolását (bár ezt még lehetne finomítani, például a perspektíva kaphatna egy saját változót a vezérsugár másolatával a subthreadben lévő vezérsugár pointere helyett).

5. Fontosabb függvények

Itt a felsorolást az alacsony sziniten ütköztetést végző függvényekkel kezdeném, a raySphereCollision-el, és a plColl-al. Ezek azért alacsony szinten ütköztetést végző függvények, mert azt mondják meg, hogy egy adott objektumnak és egy adott egyenesnek van-e metszéspontja. Ezt még sokkal tovább kell számolni ahhoz, hogy egy sok objektumból álló rendszerből értelmes képet kapjunk, de egy objektum esetén is lenne ezekkel probléma (pl a hátunk mögé is látnánk fejjel lefelé). Ezért a magas szinten ütköztetést végző függvény az engage függvény, ahol ezeknek a függvényeknek az adataival számolok tovább. Ezek a függvények egyszerűen matematikai összefüggésekből és egyenletekből adják meg a metszéspontokat, semmi egyébb trükk nincs nagyon bennük, ami pedig van, azt a forráskódban kiemeltem. A raySphereCollision esetén 2 metszéspontot kapunk vissza, ezért ennek nem jó az alap, rayColl típus, így neki jár egy egyedi visszatérési struktúra, a sphereColl. Itt a p1 és a p2 elemek a metszéspontok, a szokásos h elem pedig az ütközés bekövetkeztét jelöli. Itt p1 mindig közelebb van annak az egyenesnek a megadott pontjához, amiből raySphereCollision számol, mint p2. Az angle tagok mind a rayColl és a sphereColl típusokban az ütközés szögét jelölik. Az isMirror azt jelzi, hogy az adott objektum tükör-e, ezzel viszont csak az engage függvény fog majd számolni.

Folytatnám a sort a már sokat emlegetett engage függvénnyel. Ő legalább annyira tartozik a program szívéhez, mint az RT szálak. Ez a függvény végignézi a megadott sugár ütközését az összes mapen szereplő objektummal, és visszaadja a legközelebbi ütközési pontot. Akkor, ha az adott objektum egy tükör, akkor az ellenkező oldalon, ugyan olyan szögben a felületre nézve, létrehoz még egy sugarat, majd rekurzívan meghívja saját magát ezzel a sugárral. Ekkor a függvény visszatérési értéke a rekurzió visszatérési értéke lesz. A maximális visszatükröződést a rekurzió elindítása előtt figyelembe veszem, így nem lehet végtelen sok visszatükröződést, és végtelen hosszú rekurziót csinálni két, egymással szembefordított tükörrel (ilyen map a példák között is van).

Még szintén fontos kiemelni a calcLSAngle függvényt is. Paraméterként egy pontot kap, és visszaadja, hogy mekkora az adott pontba a fényforrásból leérkező fénysugár szöge. Ezt úgy éri el, hogy a két pontból a fényforrást választva kiindulópontnak készít egy sugarat, majd azt az engage függvénnyel ütközteti mindennel. Ha a legközelebbi ütközési pont, és a paraméterként kapott pont (nagyjából) egybe esik, akkor visszaadja az engageből származó rayColl típus szögét. Ha nem esik egybe, vagy nincs egyáltalán ütközés, akkor 0-át ad vissza.

Nagyvonalakban ezek lennének a fontosabb függvények. A különböző vektorműveleteket végző függvények általában egyszerű matematikai képleteket tartalmaznak, így nem említésre méltóak, több információ róluk a forrásban van. A többi kíírást, adatbeolvasást, stb-t végző függvény pedig nem képezi a program szerves részét, valamint egyáltalán nem tökéletesek, de nem az volt a célom, hogy egy parancsértelmező programot írjak, ahogy erre a felhasználói dokumentációban is felhívtam a figyelmet. További dokumentációt ezért ezekről nem szeretnék írni, már így is lassan 4 teljes oldalnál tartok.

6. A program néhány határa

A többi programhoz hasonlóan ez se tökéletes. Néhány nyílvánvaló hiányosság például a színek teljes hiánya, a parancsértelmezőnél pedig valószínűleg egy 5 éves indiai is jobbat írna imagine logoban. Néhány kevésbé nyílvánvaló határ azonban:

- Viszonylag sok közelítést használok, ezért nem működne a program bármilyen nagy koordináták esetén. 100000 magasságában már számítanék abnormális működésre.
- A gömb ütközései nem pontosak, ezért ha a beérekző sugár irányvektorának X (A) komponense 0, akkor ez eredményezhet a képernyő közepén egy fekete csíkot, főleg nagyon sok tükör esetén.

- Fénysugár nem tud visszatükröződni (Ezt path tracing-el lehetne szépen megcsinálni)
- Pontszerű fényforrás
- Iszonyatos erőforrásigény, és még így is alacsony képkockaszám
- Matematikailag pontatlan fényerősség számítás
- Csak a processzor kihasználása (ezt egy GPU-n nagyon szépen lehetne futtatni)
- Élsimítás hiánya (ehhez már TÉNYLEG GPU kéne)
- És minden más amihez hülye vagyok

7. Fejlesztési lehetőségek

- Objektumok mozgása: Ezt nem lenne nehéz implementálni, de nincs hozzá nagyon kedvem
- Stabil windows build: A POSIX proccessing miatt még mingw-vel se sikerült lefordítanom
- Textúrák, színek
- Nem csak matematikai egyenlettel leírható objektumok bevezetése (pl. ponthalmazok)
- Optimalizálás
- Map szerkesztő, map szerkesztése futás közben (ez se lenne nehéz, és nagyon látványos lenne)
- Több rtm (ray tracing map) fájl egyszeri megadása, és azok kombinációja

8. Ismert hibák

- Kellően sok gömb, vagy gömb visszatükröződés esetén az X = 0 irányú vektorok eltalálnak véletlenszerűen gömböket, kb tudom miért, de nem fontos kijavítani annyira, nagyon ritkán fordul elő, és ha mégis, akkor csak egy fekete vonalként jelentkezik
- Dupla tükör esetén tükör mögötti gömbök átlátszanak a tükrön egy „árnyék” szerű 2 dimenziós valami formájában