



EÖTVÖS LORÁND TUDOMÁNYEGYETEM

INFORMATIKAI KAR

ALGORITMUSOK ÉS ALKALMAZÁSAIK

TANSZÉK

Interakció fraktálokkal

Témavezető:

Bán Róbert

Doktorandusz, MSc.

Szerző:

Borbély Dávid

programtervező informatikus, BSc.

Budapest, 2020

Az eredeti szakdolgozati / diplomamunka témabejelentő helye.

Tartalomjegyzék

1. Bevezetés	2
2. Felhasználói dokumentáció	4
2.1. Felhasználói felület és funkciók	4
2.1.1. A „Parameters” feliratú panel	5
2.2. Rendszerkövetelmények és futtatás	9
3. Fejlesztői dokumentáció	10
3.1. Fejlesztői eszközök	10
3.2. Képalkotási módszer	12
3.2.1. Fénymodell	14
3.3. Megvalósítás	15
3.3.1. CMyApp osztály	15
3.3.2. gCamera osztály	19
3.4. Tesztelés	22
3.4.1. Működés helyessége	24
3.4.2. Teljesítmény	25
4. Összegzés	29
5. További fejlesztési lehetőségek	30
Irodalomjegyzék	32
Ábrajegyzék	34
Forráskódjegyzék	35

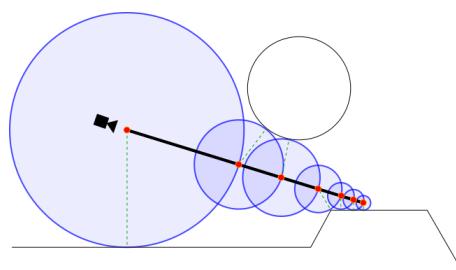
1. fejezet

Bevezetés

Ha valaki játékfejlesztésbe szeretne kezdeni, akkor nagyon sok kihívással kell szembenéznie. Szerencsére manapság már lehet egy terhet a válláról ha egy előre megírt játékmotort (**game engine**) használ, amiből akár ingyenesen elérhetőt is lehet találni.

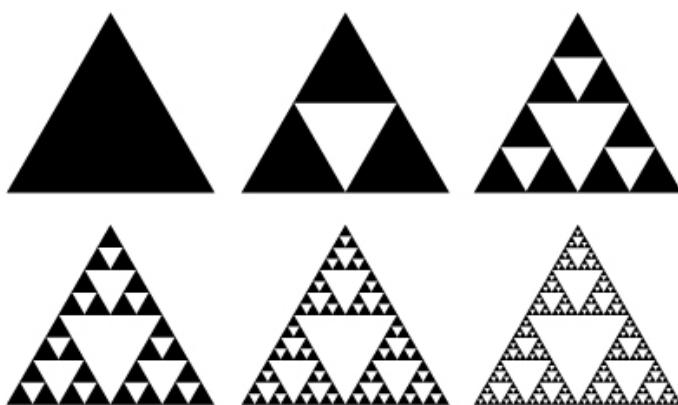
Egy játékmotor feladata hogy leegyszerűsítse a kirajzolást és az objektumok **valósághű viselkedését**. Ilyen viselkedés például, ha két szilárd tárgy ütközésekor azt várunk el hogy azok ne menjenek bele egymásba, hanem inkább ténylegesen ütközzenek és "pattanjanak le" a másikról. Ezen viselkedés kiszámolása meglehetősen költséges tud lenni, ráadásul különböző alakzatoknál különböző algoritmusokat kell használni.

Szakdolgozatomban azzal foglalkozom hogy hogyan lehet a kirajzolást és az előbb említett valósághű viselkedést fraktálokkal elvégezni. A velük való ütközéshez megállapításához ugyanaz fog segítséget nyújtani, mint a kirajzolásukhoz: távolságfüggvények.



1.1. ábra. Sphere tracing: A kamerából kiinduló fénysugár minden csakis annyit halad előre amekkora a hozzá legközelebb lévő felület távolsága [1]

Hogyan valósítom ezt meg? A kirajzoláshoz **Sphere tracing** módszert (1.1. ábra) alkalmazok, így minden kirajzolt objektumomhoz van távolságfüggvényem. Ezek segítségével meg tudom állapítani a virtuális terem bármely pontjáról hogy az milyen messze van a kirajzolt felületektől. Ezen tudással nagyon egyszerűen és hatékonyan meg lehet állapítani hogy egy gömb ütközött-e bármivel, hiszen csak annyit kell ellenőriznünk hogy a gömb középpontja gömbsugárnyi távolságra van-e valamilyen felülettől. Ezután már csak meg kell határoznunk hogy mit tegyen a gömb ütközés esetén, amihez jó kiindulási alap ha a felületről visszaverődő fénysugárként tekintünk rá - mivel annak kiszámolására jól ismert módszerek vannak.



1.2. ábra. Példa egy IFS-re: a Sierpiński háromszög néhány iterációja [2]

Ezen megközelítéshez azonban pontos és lehetőleg előjeles távolságfüggvények kellenek, így esetünkben nem érdemes foglalkozni az olyan fraktálokkal amikhez a távolságfüggvény csak felső becslést ad. Ezért olyan a fraktálok egy olyan csoportjával foglalkozom, mint a Sierpiński háromszög (1.2. ábra), amik **IFS (Iterated Function System)** által jönnek létre, azaz egy egyszerűbb alakzaton - aminek jól ismerjük a pontos távolságfüggvényét - sokszor végrehajtunk egymás után transzformációkat. Az ilyen fraktálokat könnyű generálni, mert ha eldöntöttük milyen transzformációink lesznek, azok újraparaméterezésével könnyen meghatározhatunk egy újabb fraktált leíró távolságfüggvényt.

2. fejezet

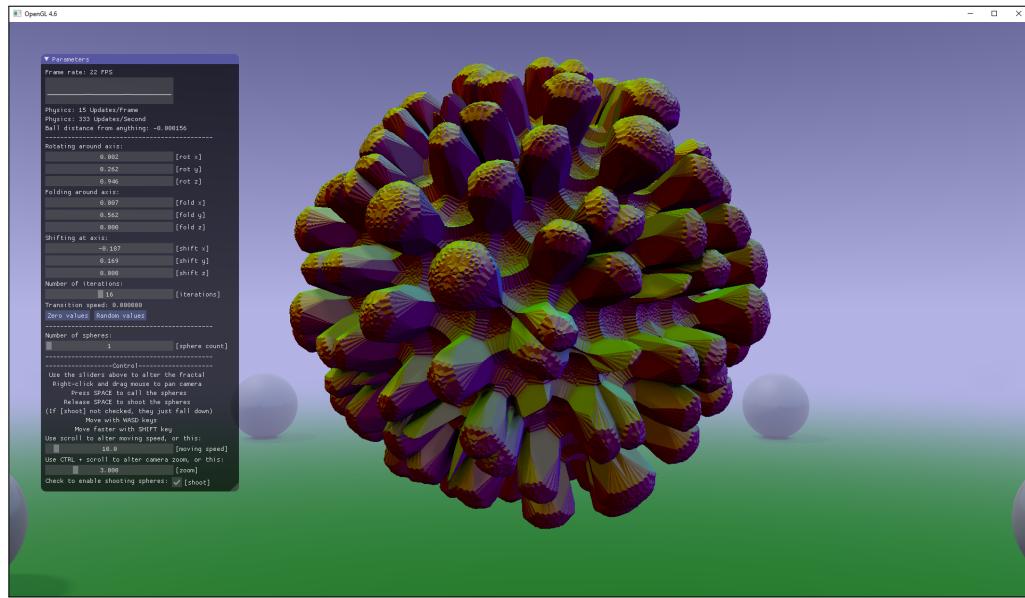
Felhasználói dokumentáció

Ezen fejezet taglalja a program futtatásához és használatához szükséges információkat. A felhasználói felület is tartalmaz rövid leírást, de a program részletes használati útmutatója a soron következő alfejezetben lesz megtalálható. A programmal egy virtuális teret lehet bezárni, melynek talaján minden irányban végtelen sok mozdíthatatlan gömb található. Van a térben továbbá egy nem aktívan mozgó, de testre szabható fraktálunk, valamint vannak mindenfelé kilőhető és mindenről visszapattanó labdák, melyek a programban megírt fizika alapján viselkednek.

2.1. Felhasználói felület és funkciók

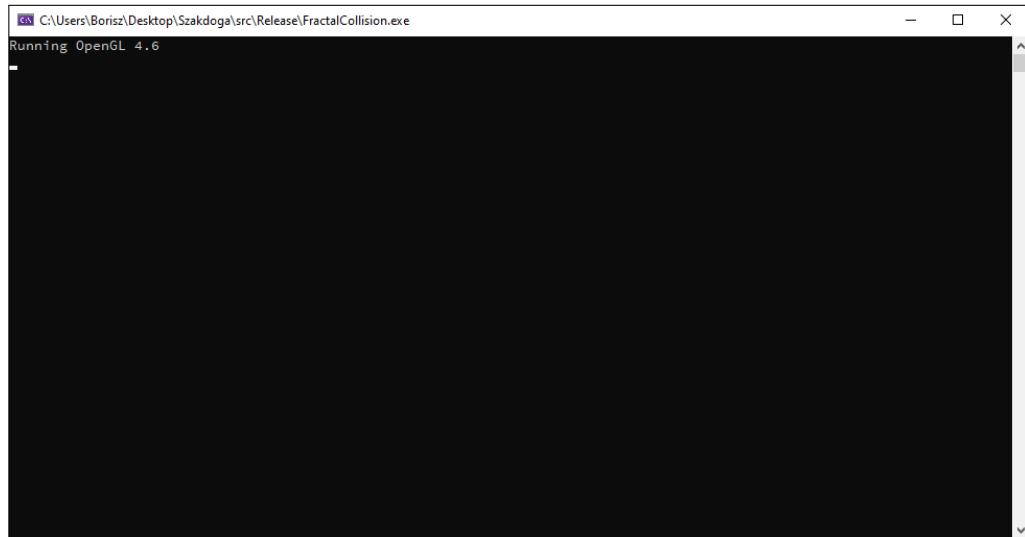
A program sikeres indítása után – melyről a 2.2. fejezetben tudhatunk meg többet – kettő darab ablakkal találjuk szembe magunkat. Ezekről a 2.1. és 2.2. ábrák mutatnak egy-egy képernyőképet.

A 2.1. ábrán látható ablak tartalmazza a program lényeges részét, itt jelenik meg a kirajzolt képünk és ebben az ablakban található a „**Parameters**” feliratú panel, melynek segítségével különböző paramétereket tudunk nyomon követni és módosítani. Az ablak alapértelmezetten 1920x1080 nagyságú, de szabadon átméretezhető, viszont az ablak mérete befolyással van a teljesítményre! Mindig az ablak pontos felbontásában fog renderelni, így nem optimális teljesítmény esetén érdemes megfontolni az ablak kisebbre vételét.



2.1. ábra. Fő programablak

A 2.2. ábrán vehetjük szemügyre azt a terminálablakot mely az esetleges hibaüzeneteket fogja kiírni. Ezen kívül ez az ablak más funkcióval nem bír.

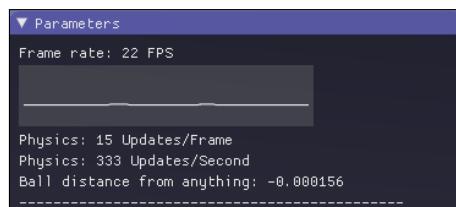


2.2. ábra. Terminálablak

2.1.1. A „Parameters” feliratú panel

A „Parameters” feliratú panel nagy jelentőséggel bír, így a jobb olvashatóság végett nem csak a 2.1. ábra részeként láthatjuk hanem külön is szerepel a 2.3., 2.4. és 2.5. ábrákon.

Ezen a panelen számos információt tudhatunk meg és állíthatunk át a program futásával kapcsolatosan. Alapértelmezetten a fő programablak bal oldalán található, de szabadon mozgatható és átméretezhető az ablakon belül, indításkor pedig az legutóbbi futtatás végén beállított pozíciót és méretet veszi fel.

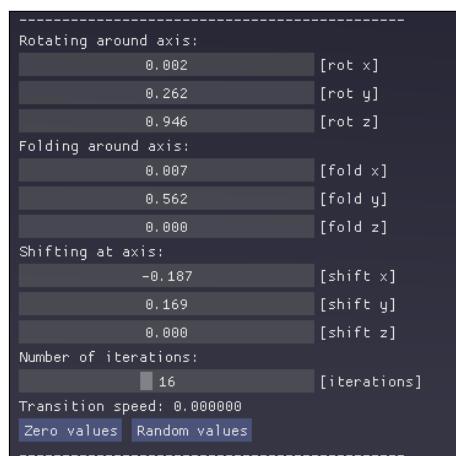


2.3. ábra. A „Parameters” feliratú panel felső harmada

A panel legtetején (2.3. ábra) találhatjuk a „**Frame rate:**” felirat után az aktuális képfrissítési rátát képkocka/másodperc mértékegységben, illetve közvetlenül ezalatt az utolsó másfél másodperc adatait követhetjük nyomon egy folyamatosan frissülő grafikonon.

A két „**Physics:**” felirat után olvashatjuk le hogy milyen gyakran van a labdák mozgása frissítve frissítés/képkocka és frissítés/másodperc mértékegységekben. Egy frissítés során minden labda sebessége és pozíciója újraszámolódik, valamint ellenőrzésre kerül az is hogy ütközött-e valamivel.

A „**Ball distance from anything:**” felirat után olvasható a dobálható labda távolsága a tőle legközelebb lévő felülettől - több labda esetén az utoljára létrehozotttra vonatkozik. Az apró ingadozásából látszik hogy igazából folyamatosan pattog a labda, csak ez a pattogás egy idő után elhanyagolható mértékű lesz.



2.4. ábra. A "Parameters" feliratú panel középső harmada

A választóvonal alatti szekcióban (2.4. ábra) a fraktálunkat tudjuk személyre szabni. A fraktálunk úgy rajzolódik ki hogy egy 1x1x2 egység nagyságú téglatesten egymás után többször végrehajtunk különböző transzformációkat. Ezen transzformációk paramétereit tudjuk beállítani a következő 9 db határ nélküli csúszkán - mely ugyanúgy működik mint egy sima csúszka, csak nincsen minimum és maximum értéke és az egeret tovább is lehet húzni mint a csúszka vége. Mindegyik csúszkának CTRL + kattintással begépelet értéket is meg lehet adni.

A **[rot x]**, **[rot y]**, **[rot z]** csúszkákkal azt tudjuk szabályozni hogy mennyire legyen elforgatva egy iterációban a fraktál az X, Y és Z tengelyek körül (radiánban értendők az értékek).

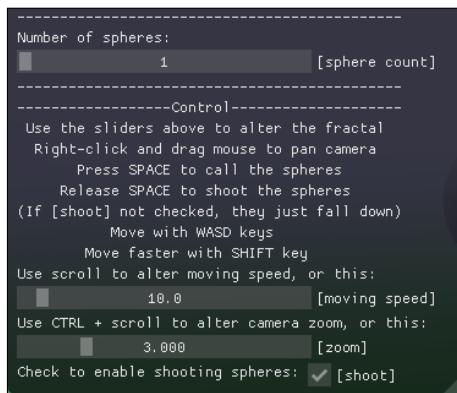
A **[fold x]**, **[fold y]**, **[fold z]** csúszkákkal azt tudjuk szabályozni hogy mennyire legyen elforgatva az adott tengely körül a tükrözősík ami a megadott szöggel (radián) fordul el és tükrözi az alakzatot iterációinkként. Itt a három érték 3 különböző tükrözősíkot forgat el a nevükben szereplő tengely mentén.

A **[shift x]**, **[shift y]**, **[shift z]** csúszkák szabályozzák hogy mennyire legyen eltolva az alakzat iterációinként az X, Y és Z tengely mentén.

Végül pedig az **[iterations]** csúszka, amely már korlátozva van az [1,36] tartományban, beállítja hogy az előző 9 csúszka által paraméterezett 9 transzformáció hányszor legyen végrehajtva a téglatesten. Ez a paraméter van a legnagyobb hatással a futás sebességére, így gyengébb gépeken nem érdemes nagy értéket beállítani. Ha a képfrissítési ráta 10 képkocka/másodperc alá csökken akkor automatikusan elkezd csökkenni a csúszka értéke.

Itt található még két gomb: a „**Zero values**”, mely a fraktál paramétereit nullához közelíti¹, illetve a „**Random values**”, mely véletlenszerű értékeket állít be ezeknek. Az iterációk számát egyik sem állítja át. Továbbá van még egy ezekhez szorosan kacsolódó érték ami a „**Transition speed:**” felirat után olvasható. A gombok által generált új paramétereket egy 5 másodperc hosszú fázisban folyamatosan, apránként közelíti az aktuális paraméterekkel, az előbb említett érték pedig azt fejezi ki hogy milyen súlyozással veszi az aktuális és a célérték átlagát.

¹Az értékek csak konvergálnak a 0-hoz. Ha valamely paraméter szélsőségesen nagy, előfordulhat hogy a gomb megnyomása után is jelentősen eltér nullától



2.5. ábra. A "Parameters" feliratú panel alsó harmada

Az alsó harmadban (2.5. ábra) található a **[sphere count]** csúszka, itt 1 és 100 között lehet értékeket beállítani. Ez is jelentősen befolyásolja a futás gyorsaságát, így 15 FPS alatt ez az érték is automatikusan csökken.

Van egy rövid ismertető szövege a panelnek, ami azt a célt szolgálja hogy ezen dokumentáció nélkül is tudja használni egy felhasználó ha leül a program elé. Ebben kerülnek ismertetésre a virtuális tér bezárásához szükséges irányítások is. A mozgás-hoz a **WASD** billentyűket és az egeret lenyomott jobb egérgombbal² kell használni a legtöbb játékban megszokott módon – a kamera szabadon repül bármilyen irányba. A mozgási sebességet a **[moving speed]** csúszkával lehet személyre szabni, illetve ugyanezen csúszka értékét az **egérgörgővel** is lehet szabályozni. A **shift** billentyű lenyomására ideiglenesen megnégyszereződik a sebesség, felengedésére visszaáll. A kamera látószögét lehet csökkenteni a **[zoom]** csúszka értékének növelésével, vagy a **Ctrl + görögő** segítségével is.

A labdákat a **szóköz** billentyű nyomva tartásával lehet magunkhoz hívni. Egy labda esetén az ablak közepére, több labda esetén a labdák az ablak közepe körül keringenek egy körvonallal mentén egyenletesen elhelyezkedve. A szóköz billentyűt felengedve egyszerre kilövődnek a labdák, ha be van pipálva a panel alján a **[shoot]** jelölőnégyzet, ha nincsen bepipálva csak leesnek a gravitációtól megfelelően.

²Erre azért volt szükség mert a paraméterek beállításához is az egeret kell használni, muszáj volt elkülöníteni valahogy a kettőt

2.2. Rendszerkövetelmények és futtatás

Az alkalmazás üzembe helyezésének követelménye a Windows 7 vagy afeletti operációs rendszer és az OpenGL 3.0 (vagy afeletti verzió) hardveres támogatása. Azonban az alkalmazás rettentően GPU intenzív, így az optimális futáshoz elengedhetetlen a dedikált videokártya. A teszteléshez használt számítógép specifikációi:

- Intel® Core™ i7-8700 CPU
- 16 GB RAM
- NVIDIA GeForce GTX 1660 GPU
- Windows 10 operációs rendszer

Ezen konfigurációval, 1920x1080 felbontás mellett a program sebessége elfogadható volt.

Az alkalmazás elindításához a **FractalCollision.exe** fájlt kell futtatni. Fontos hogy az exe fájl mellett ott legyen a **shader.frag** és a **shader.vert** fájlok, illetve ha a rendszeren nincsen külön telepítve akkor az **SDL2.dll**, a **glew32.dll**, a **vcruntime140.dll** és a **msvcp140.dll** fájloknak is az exe mellett kell lenniük. Ezek mind a **Release** mappában vannak, így onnan indítva erre nem kell ügyelni.

Az alkalmazásból való kilépéshez lehet az **Esc** billentyűt vagy a jobb felső sarkban az ablak bezárás gombját használni. Ha bezárnak a terminálablakot akkor minden ablak bezárul, ha először a fő programablakot zárjuk be akkor utána még külön be kell zárni a terminálablakot.

3. fejezet

Fejlesztői dokumentáció

Az alkalmazás **Microsoft Visual Studio** segítségével készült. A kódok elsősorban C++, másodsorban – a shaderek – GLSL nyelven íródtak. Ha újra akarnánk fordítani akkor a **C:/** helyre csomagoljuk ki a mellékelt OGLPack.zip állományt (ez az szükséges csomagokat és függőségeket tartalmazza), majd futtassuk a **subst T: C:/** parancsot. Ezután megnyithatjuk a **.vcxproj** projektfájlt.

3.1. Fejlesztői eszközök

A program írása során számos könyvtár és API felhasználásra került. Ebben az alfejezetben ezek kerülnek bemutatásra.

Simple DirectMedia Layer (SDL)

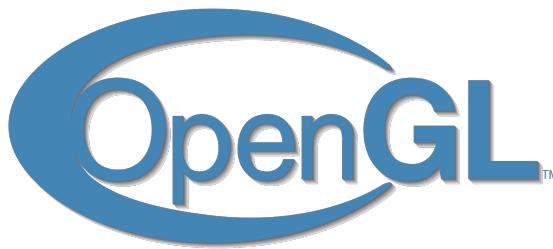
A Simple DirectMedia Layer (SDL) könyvtár egy crossplatform multimédiás könyvtár, ami alacsony szintű, hatékony hozzáférést ad audio, bemeneti (egér, billentyűzet, joystick), valamint grafikus (OpenGL-en keresztül GPU-hoz) eszközökhöz. Az alkalmazásban az SDL 2.0 van használva. [3]



3.1. ábra. Az SDL logója

OpenGL API

Az OpenGL (Open Graphics Library) egy részletesen kidolgozott szabvány, melyet a Silicon Graphics nevű amerikai cég fejlesztett ki 1992-ben. Olyan API-t takar, amely segítségével egy egyszerű, szabványos felületen keresztül megvalósítható a grafikus kártya kezelése és a háromdimenziós grafika programozása. Az interfész több ezer különböző függvényhívásból áll, melynek segítségével a programozók szinte közvetlenül vezérelhetik a grafikus kártyát, segítségükkel 3 dimenziós alakzatokat rajzolhatnak ki, és a kirajzolás módját szabályozhatják. [4]



3.2. ábra. Az OpenGL logója

Dear ImGui

A „Parameters” feliartú panel megjelenítése ennek segítségével lett megoldva. A Dear ImGui (ImGui) egy egyszerű grafikai interfész könyvtár C++ nyelvhez. Segítségével egyszerűen kezelhetünk gombokat, csúszkákat, egyéb beviteli eszközöket valamint könnyen megjeleníthetünk adatokat. Gyors és hordozható, nincs szükség külső könyvtára csak néhány szimpla forrás fájl beillesztésére. [5]

GLM

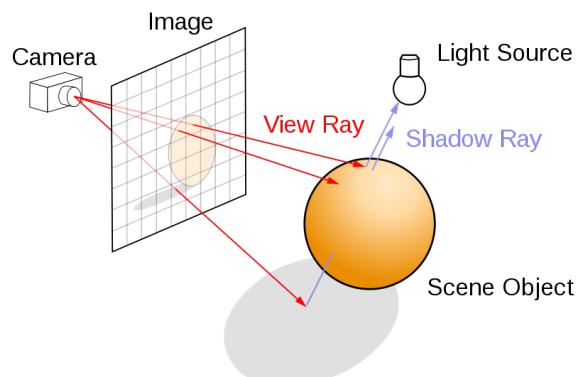
Az OpenGL Mathematics egy headerben importálható C++ könyvtár mely a GLSL specifikációira szabva (elnevezések, funkciók) tartalmaz számtalan matematikai funkciót. Segítségével a GLSL-ben használatos funkciók és típusok C++-ban is használhatóak. [6]



3.3. ábra. A GLM logója

3.2. Képalkotási módszer

A megjelenítés **Raycast** technika (3.4. ábra) alkalmazásával történik, egy speciális implicit reprezentáción, a távolságfüggvényeken. Ehhez minden pixelre ki kell számolni egy sugár paramétereit. Ezen sugár és felület metszetét a **Sphere tracing** algoritmussal kapjuk meg. A felületi normálist numerikusan számítjuk ki, melynek segítségével a felület már könnyen árnyalható.



3.4. ábra. A raycast technika ábrázolása. [7]

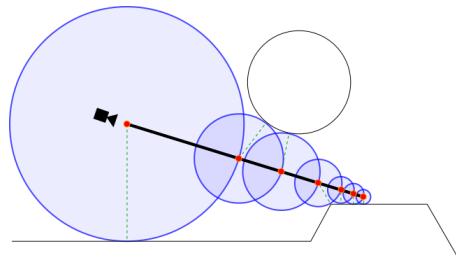
A **sphere tracing** algoritmus egy speciális esete az úgynevezett **raymarching** algoritmusnak. Működéséhez a kirajzolni kívánt objektumokat távolságfüggvényekkel kell reprezentálni, mely a virtuális tér bármely pontjáról képes meghatározni hogy milyen messze van az objektum felületétől. Az algoritmus egy implementációja a 3.1. forráskód részletben látható.

```

1 float RayMarch(vec3 ro, vec3 rd) {
2     float dist=0.0;
3     for(int i=0; i<MAX_STEPS; i++) {
4         vec3 p = ro + rd*dist;
5         float dS = GetDist(p);
6         dist += dS;
7         if(dist>MAX_DIST || dS<SURF_DIST ) break;
8     }
9     return dist;
10 }
```

3.1. forráskód. A sphere tracing algoritmust megvalósító kód

Az algoritmus két paramétert igényel: egy pontot és egy vektort, melyek meghatározzák a sugár kezdőpontját és irányát. Ezen irány mentén lépked folyamatosan minden annyit amennyit amekkora a távolság a legközelebbi felülethez képest. Ezt addig csinálja míg már kellően közel lesz ($dS < SURF_DIST$), vagy ha már túl messzire ment ($dist > MAX_DIST$), vagy esetleg túl sokat lépett ($i > MAX_STEP$).



3.5. ábra. Sphere tracing: A kamerából kiinduló fénysugár mindenkorán halad előre amekkora a hozzá legközelebb lévő felület távolsága [1]

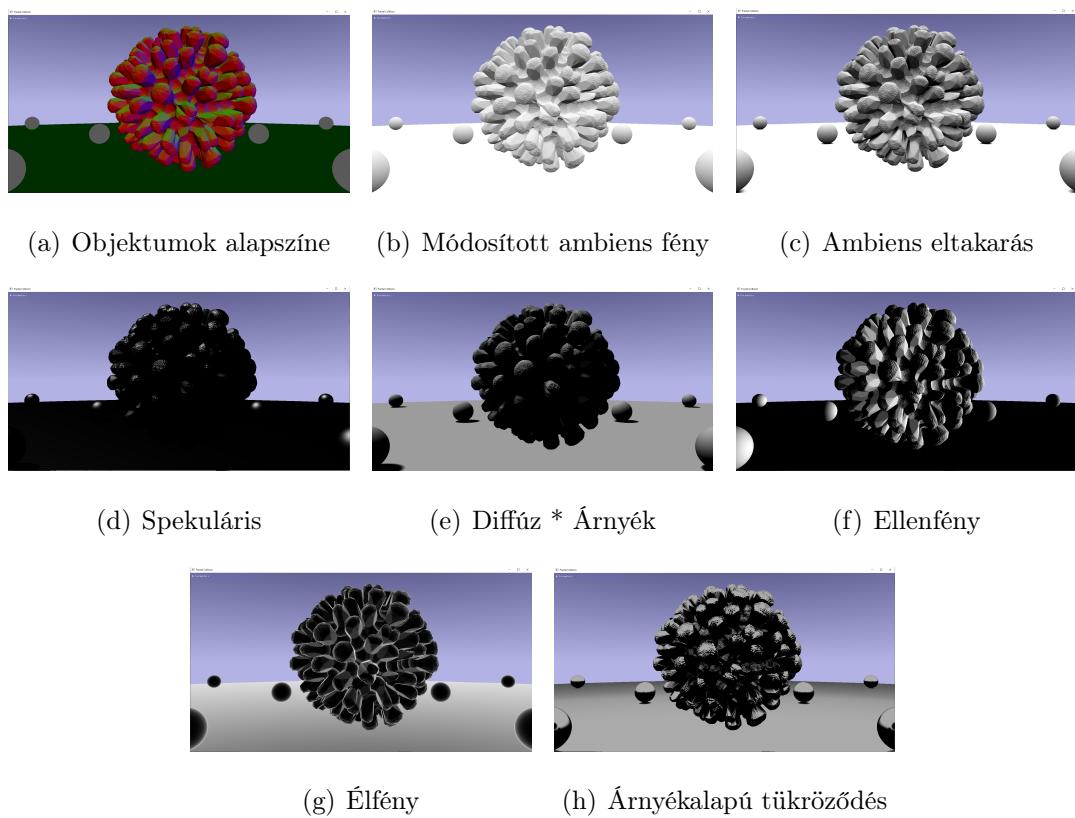
Az alkalmazásban teljesítményjavítás érdekében másik feltétel ($dS < 0.01 * dist / MAX_DIST$) lett használva a felületi távolsághoz, mely számításba veszi hogy mennyire messzire vagyunk a kamerától. Ha messzebb vagyunk

akkor nincsen szükség akkora részletességre és a felületi távolság így lehet nagyobb is.

3.2.1. Fénymodell

Ha túl messzire ment a sugarunk ($\text{dist} > \text{MAX_DIST}$), akkor egyszerűen a háttér színét adjuk a pixelnek. Egyéb esetben pedig kiszámoljuk az adott pontban a felületi normálist és a fénymodell segítségével megállapítjuk a pixel színét.

A használt fénymodell nagyon sokat számít a kirajzolt kép minőségén. Ezért sok idő és energia lett rászánva ennek megalkotására, a végeredményre Ingio Quilez munkássága [8] nagy befolyást gyakorlot. Ezen fénymodell komponensei a 3.6. ábrán láthatóak. Ezek különböző színenkénti súlyozással vett összege alkotja végső színarányalatot, melyre még egy gamma korrekció és egy ködszerű effektus is került – ez utóbbi arra szolgál hogy a viszonylag kicsi maximális távolság leplezze.



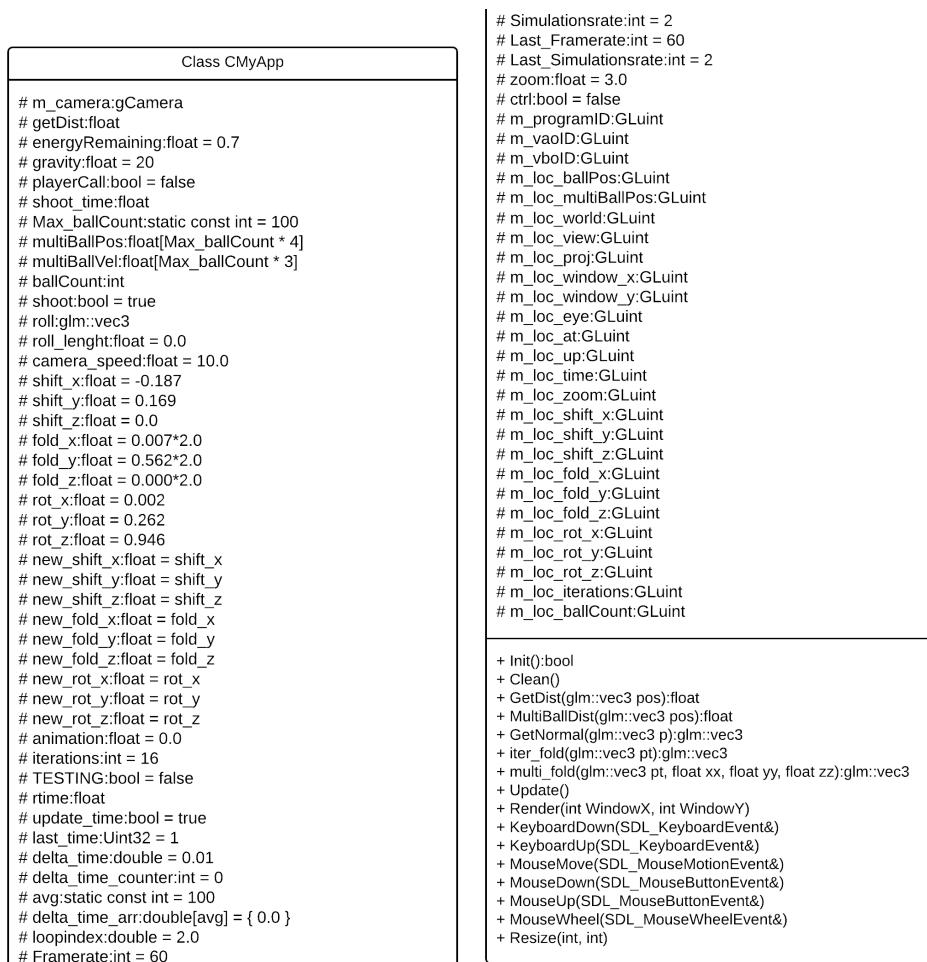
3.6. ábra. A fénymodell különböző komponensei.

3.3. Megvalósítás

A program írása során a Számítógépes Grafika BSc gyakorlat tárgy honlapjának [9] projektjei nyújtottak kiindulási alapot. Az SDL működtetése és számos alapvető függvényhívás lett belőlük felhasználva.

3.3.1. CMyApp osztály

Ez a fő osztály. A main.cpp ezen keresztül kezeli le az egér és billentyűzet bemeneiteit, szimulálja a fizikát és továbbítja a shaderbe a paramétereket hogy megtörténjen a kirajzolás. A fontosabb függvények külön részletezésre kerülnek.



(a) A diagram felső fele

(b) A diagram alsó fele

3.7. ábra. A CMyApp osztálydiagramja (két részre szedve a hosszúsága miatt)

bool Init()

Ebben inicializálódik minden aminek kell. Definiálunk két darab háromszöget melyek összerakva egy olyan négyzetlapot alkotnak ami az XY síkot (-1,-1)-től (1,1)-ig lefedik. Ezt a négyzetlapot fogjuk később a shaderben átszínezni hogy megkapjuk a képet. Az uniform változók memóriacíméit is itt határozzuk meg, valamint a mozgatható labdák pozícióját, méretét és sebességét tartalmazó tömbök is itt kapnak kezdőértékeket.

void Update()

Minden képernyőfrissítés előtt lefut, ebben történik a ütközések ellenőrzése és a labdák mozgatása a pozícióik frissítése által.

A labdák befogott pozíciói – amikhez akkor közelítenek ha lenyomjuk a szóköz billentyűt – itt kerülnek meghatározásra a kamerát leíró adatok ismeretében. Meg tudunk határozni egy a kamerából előre, egy jobbra és egy felfelé irányba mutató egységhosszú vektort. Ezek megfelelő kombinációjával és egy időtől és a labdák darabszámtól függő forgatási mátrix felhasználásával tudjuk a pozíciójukat egy a kamerához képest fix helyzetű kör mentén beállítani. Mivel az időtől is függ a forgatási mátrix, így ezen kör mentén folyamatosan keringnek.

A labdák hívása és kilövése is itt van megvalósítva. A **SPACE** nyomva tartásakor labdák az imént kiszámolt pozíciójának és a jelenlegi pozíciójának különbségének konstans-szorosát kapja meg sebességiül. Ezáltal minél messzebb van a pozíciótól annál gyorsabban közeledik hozzá. Illetve csak a billentyű nyomva tartásakor frissül egy **shoot_time** változóban az idő.

Ha az aktuális idő és a **shoot_time** közötti idő csak kis mértékben tér el akkor tudjuk hogy most lett felengedve a billentyű. Ekkor pedig a kamerából előrefelé mutató vektor konstans-szorosa adódik a labdák sebességéhez.

A labdák **ütközéseinek megállapításához** a kirajzoláshoz is használt távolságfüggvényt használjuk a vizsgált labda középpontjával. Ha ez a távolság kisebb mint a gömb sugara akkor tudjuk valamivel ütköztünk.

A helyes **viselkedés megállapításához** ismerünk kell az ütközés pontjában a felületi normálist. A normális kiszámolásához használt módszer (3.2. kód részlet) a távolságfüggvényt használja így egy egyszerűsítést tehetünk: az ütközési pont helyett

a gömb középpontjában számoljuk a felületi normálist! Ezáltal az éleken és sarkokon ahol felületi normális nem igazán értelmezhető, ott is jó viselkedést produkáló vektort fogjuk kapni.

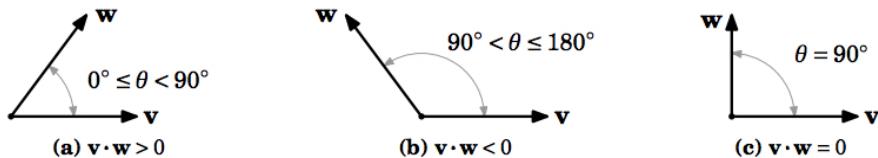
```

1 glm::vec3 CMyApp::GetNormal(glm::vec3 p) {
2     float d = GetDist(p);
3     float e = 0.0005;
4     glm::vec3 n = d - glm::vec3(
5         GetDist(p - glm::vec3(e, 0, 0)),
6         GetDist(p - glm::vec3(0, e, 0)),
7         GetDist(p - glm::vec3(0, 0, e)));
8     return glm::normalize(n);
9 }
```

3.2. forráskód. A felületi normálist kiszámoló függvény

Ezután már csak a kapott normálisra kell tükrözniünk a vizsgált labdánk sebességvektorát és a megfelelő komponenseit némi leg csökkenteni, ezzel szimulálva hogy visszapattanáskor veszít egy keveset az energiából – ehhez is a normálist használhatjuk, azért kell komponensenként mert a valóságban ha például elrúgunk ívesen egy focilabdát, akkor annak az vízszintes irányú mozgási energiája jóval kevesebbet csökken visszapattanáskor mint a függőleges irányú.

Fel kell arra is készülnünk ha a **labdánk belemegy egy másik objektumba**. Ez többnyire akkor történik meg ha nagyon gyorsan mozog a labda és két ellenőrzés között beleér, vagy ha a labda tartásakor direkt belevíssük egy objektumba. Ha benne vagyunk valamiben akkor frissítjük a pozíció értékét a normális irányába egy kis mértékben. Ezáltal ha esetleg belekerülne valamibe a labdánk akkor kijön belőle automatikusan.



3.8. ábra. A skaláris szorzat előjele és a vektorok közötti szög összefüggése [10]

Mivel tudjuk hogy akár egy másik objektumba is belemehet a labdánk ezért még egy esetre fel kell készülnünk: a másik objektumból kifelé jövet nem szeretnénk hogy

ismét tükröződjön a sebességek vektor, hiszen akkor állandóan oda-vissza tükröződne és sosem jutna ki a labda. Erre megoldás hogy csak akkor tükrözzük a sebességeket ha a normálvektorral bezárt szöge **tompaszög**. Ehhez minden össze a két vektor skaláris szorzatát kell venni és ellenőrizni hogy az eredmény negatív-e.

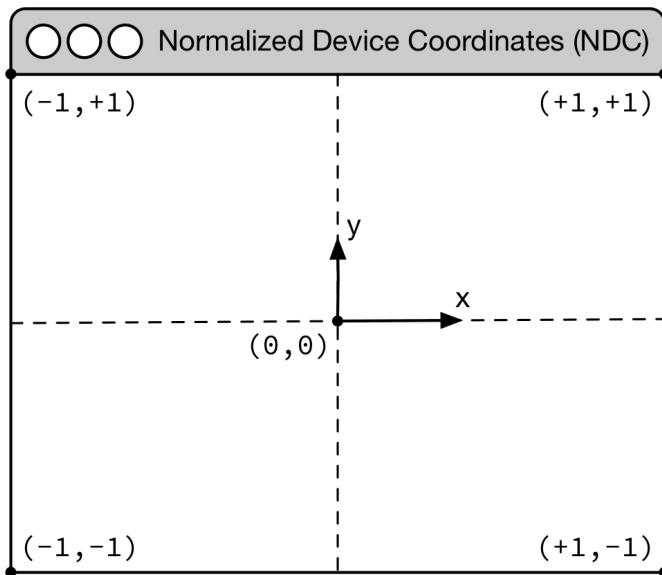
void Render(int WindowX, int WindowY)

Paraméterként megkapja az ablak méreteit. Az imgui panel ebben a függvényben hívódik meg és töltődik fel a megjelenített tartalommal, illetve itt kapják meg a felületen bevitt értékeket a megfelelő változók. Itt hívódik meg továbbá a shader is, melynek rengeteg uniform változót adunk át:

- **WindowX, WindowY** - az ablak méretei
- **eye, at, up** - a kamerát definiáló vektorok
- **rtime** - az indítás óta eltelt idő másodpercben
- **multiBallPos** - a mozgó labdák koordinátáit és méreteit tartalmazó **tömb**
- **shift_x, shift_y, shift_z** - az eltolás transzfomácó paraméterei
- **fold_x, fold_y, fold_z** - a tükrözés transformációk paraméterei
- **rot_x, rot_y, rot_z** - a forgatás traszformáció paraméterei
- **iterations** - a fraktál iterációinak száma
- **ballCount** - a mozgó labdák száma
- **zoom** - a kamera látószögét befolyásoló paraméter

A kép megalkotása teljes egészében a shaderben történik. Miután megtörtént az imgui panel konfigurásála és az uniform változók átadása, a **glDrawArrays()** függvény hívásával a vertex shaderbe juttatjuk az inicializáció során létrehozott és a futás során sehol nem módosított négyzetet.

Kezdetben egy négyzetet adtunk meg és ezt ablak formájúra kell nyújtani. A vertex shaderben az ablak méreteinek ismeretében kompenzáljuk a torzítást, azáltal hogy a továbbadott vektor x komponensét megsorozzuk az ablak szélességének és magasságának hányadosával.



3.9. ábra. NDC (Normalized Device Coordinates) - az ablak koordinátáit normalizáltuk, hogy a ball alsó sarok (-1,-1), a jobb felső sarok (+1,+1) legyen. [11]

Ennek köszönhetően a fragment shaderben a bemeneti vektorunk egy olyan leképezés eredménye, ami az ablak minden pixeléhez hozzárendel egy -1 és 1 közötti (X,Y) koordinátapárost ahogy az a 3.9. ábrán is látható. Ezután a fragment sahderben a 3.2. fejezetben ismertetett elmélet alapján megalkotársa kerül a kép.

3.3.2. gCamera osztály

Ez az osztály egy általános megvalósítása egy virtuális kamerának és közel egy-az egyben lett felhasználva a Számítógépes Grafika BSc gyakorlat tárgy honlapjának [9] projektjeiből.

Csak a kamerát mozgató függvényekre és az általuk befolyásolt eye, at és up vektorokra van szükség. Ezen három vektor segítségével lehet majd a shaderben megállapítani a sugarak irányát és kiindulási pontját.



3.10. ábra. A gCamera osztálydiagramma

Ebben az osztályban valósul meg a virtuális kamera mozgatása, hiszen a helyváltoztatáshoz mindenhez a kamerát leíró vektorokat kell megfelelően transzformálni. Az ehhez szükséges függvények kerülnek részletezésre.

void Update(float _deltaTime)

Ez a függvény minden kirajzolás előtt meghívódik és frissíti a kamerát definiáló vektorokat. Ehhez ismernünk kell az előre (**m_fw**) és az oldalra mutató (**m_st**) vektorokat melyeket a megfelelő szorzóval egyszerűen hozzáadunk az **eye** és **at** vektorainkhoz (**m_eye** és **m_at**) ahogyan azt a 3.3. kódrészletben is láthatjuk.

```

1   m_eye += (m_goFw*m_fw + m_goRight*m_st)*m_speed*_deltaTime;
2   m_at  += (m_goFw*m_fw + m_goRight*m_st)*m_speed*_deltaTime;

```

3.3. forráskód. Az **eye** és **at** vektorok frissítése

A szorzókat (**m_goFw** és **m_goRight**) a billentyűk fogják szabályozni. Megfigyelhetjük ha ezen változók értéke +1 vagy -1 akkor előre-hátra és jobbra-balra is tudunk menni. Ellenben ha minden kettő nulla, akkor nem változnak az **eye** és **at** vektoriink, vagyis ekkor nem fog mozogni a virtuális kamera.

Látható továbbá hogy a mozgás mértéke az **m_speed** változótól és a **_deltaTime-tól** fognak függeni. A felületen az **m_speed** értékét állítjuk át az osztály **SetSpeed(float _val)** függvényen keresztül, ezzel szabályozván a mozgási sebességet.

void KeyboardDown(SDL_KeyboardEvent& key)

A **WASD** és a **SHIFT** billentyűk segítségével tudunk mozogni a térben ezért ezeket a billentyűket folyamatason figyelni kell hogy le vannak-e nyomva. Ez egy switch és a paraméterül kapott esemény segítségével történik meg.

A **W** és **S** billentyűk lenyomására az **m_goFw** változó értéke rendre 1 és -1 értéket kap. Az **A** és **D** billentyűk hatására pedig hasonlóképpen az **m_goRight** változónak adunk 1 és -1 értéket.

A **SHIFT** billentyű lenyomására, ha az **m_slow** változó éréke hamis, akkor az **m_speed** változó értékét növeljük a négyeszeresére, illetve az az **m_slow-t** igazra állítjuk. Azért szükséges ezen változó figyelése mert csak egyszer szeretnénk meg-négyezni a sebességet.

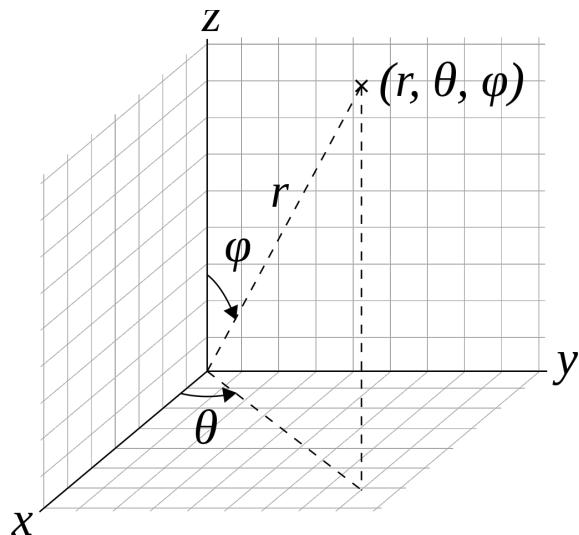
void KeyboardUp(SDL_KeyboardEvent& key)

A billentyűk felengedését külön kell kezelni, ez ugyanolyan elven történik mint a **KeyboardDown()** esetében: A **W** és **S** billentyűk felengedésére az **m_goFw** változó értéke 0 lesz. Az **A** és **D** billentyűk felengedésére pedig az **m_goRight** változót nullázzuk.

A **SHIFT** billentyű felengedésére, ha az **m_slow** változó éréke igaz, akkor az **m_speed** változó értékét változtatjuk a negyedére, valamint az **m_slow** hamis lesz.

void UpdateUV(float du, float dv)

Ez a függvény a 3.3. kódrészletben is látható **m_fw** és **m_st**, vagyis az előre és oladra mutató vektoroknak fog megfelelő értéket adni és frissíti az **m_at** vektorunkat. Az osztály **MouseMove(SDL_MouseMotionEvent& mouse)** függvénye fogja megadni neki a **du** és **dv** paramétereket az egér függőleges és vízszintes elmozdulása alapján. Ez is minden kirajzolás előtt meghívódik.



3.11. ábra. Polár koordinák [12]

Az **m_at** vektorunkat (mely azt határozza meg hogy merre néz a kamera) leírjuk polárkoordinátákkal és a megfelelő szögekhez hozzáadjuk – az értelmezési tartományok figyelembe vételevel – a paraméterül kapott **du** és **dv** értékeket. Ezután átszámoljuk az így kapott vektort Descartes-féle koordinátáakra és ez lesz a **m_at** vektorunk új értéke.

Az **m_fw** vetort az **m_at** és **m_eye** különbsége fogja adni, az **m_st** vetort pedig a **m_fw** és **m_up** vektoriális szorzata.

3.4. Tesztelés

A tesztelés során megvizsgáltam hogy a funkciók (2.1. fejezet) az elvártaknak megfelelően működne-e. Az ehhez használt számítógép konfiguráció:

- Intel® Core™ i7-8700 CPU
- 16 GB RAM

- NVIDIA GeForce GTX 1660 GPU
- Windows 10 operációs rendszer

3.4.1. Működés helyessége

A következő viselkedéseket ellenőriztem:

1. A virtuális kamerát lehet forgatni az **egérrel**;
2. A virtuális kamera nagyítását lehet állítani a **CTRL + görgővel** és a csúszkával is;
3. A virtuális kamerát lehet mozgatni a **WASD** billentyűkkel;
4. A mozgás sebességét lehet gyorsítani a **SHIFT** billentyűvel és állítani a görögővel vagy a csúszkával;
5. Ha egyszerre gyorsítunk **SHIFT**-tel és görögővel, a görög sebessége felülírja a shift gyorsítását;
6. A fraktál paramétereit át lehet állítani a **csúszkákkal** és a fraktál ezen értékkeknek megfelelően változik;
7. A „**Zero values**” gomb hatására megközelíti minden érték a nullát;
8. A „**Zero values**” gomb extrém érték esetén: 10000-re állítva egy csúszkát "nullázás" után az értéke 0.481 lett, 36-os iteráció mellett;
9. A „**Random values**” gomb valóban véletlenszerű értékeket állít be. Időnként nem látható fraktálokat eredményez, minél nagyobb az iterácó értéke annál gyakrabban;
10. A **SPACE** billentyű nyomva tartásakor a labdák megjelennek a felhasználó előtt. Ha közeli akadályba ütköztek akkor nem tudnak;
11. A **SPACE** billentyű felengedésekor a labdák kilövődnek ha be van pipálva a **[shoot]**, és leesnek a gravitációt megfelelően ha nincsen;
12. A **labdák** ütközése tárgyakkal valósághűnek tűnik;
13. A **labdák** ütközése egymással közepesen valóságosnak tűnik, a kezdőpozíció alatti félgömbhéjba folyamatosan eltolják egymást.
14. A **labdák** legurulása lejtős felületeken lassabb mint kellene, közepesen valósághű.
15. A **labdák** legurulása lejtős felületeken nem egyenletes, bizonyos irányú lejtők kevésbé működnek jól.

3.4.2. Teljesítmény

Az alkalmazás erősen GPU igényes. A futás teljesítményét az alábbi kód segít-ségével teszteltem:

```

1 if (TESTING)
2 {
3     if (delta_time_counter < avg)
4     {
5         delta_time_arr[delta_time_counter] = delta_time;
6         ++delta_time_counter;
7     }
8     else
9     {
10        delta_time_counter = 0;
11        double avg_delta_time = 0.0;
12        for (int i = 0; i < avg; ++i) { avg_delta_time +=
13            delta_time_arr[i]; }
14        avg_delta_time /= avg;
15        printf("Averge of delta time: %f ms    Iterations: %d    Number
16          of spheres: %d \n", avg_delta_time*1000, iterations,
17          ballCount);
18        iterations += 2;
19    }
20 }
```

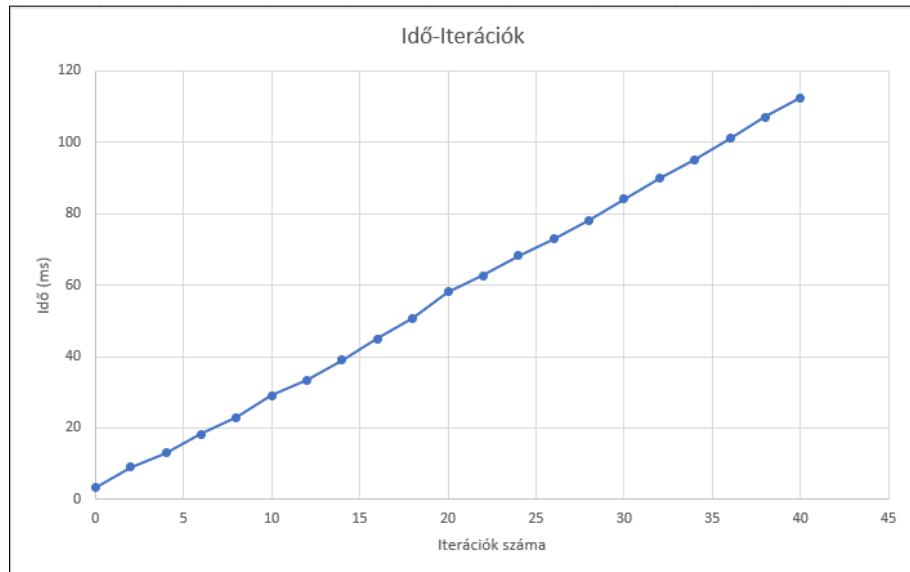
3.4. forráskód. A tesztelést végző kód

A kód az **Update()** függvény alján található. A **delta_time** két képfrissítés között eltelt időt jelöli. Ezen kód segítségével **avg** darab képfrissítési idő átlagát vesszük, ezt kiírjuk az aktuális iterációk száma és mozgatható gömbök száma mellett a terminálablakra, majd ez előbbi értékét megnöveljük kettővel. A tesztek avg=100 értékkel futottak.

A tesztelés idejére ki lett kapcsolva a **vsync**, hogy ne befolyásolja a mérést. Ha be lenne kapcsolva akkor az $1/60 \text{ s} = 16.66 \text{ ms}$ -nál kisebb kirajzolási idők eredményét nem tudnánk lemérni. Továbbá az a funkció is ideiglenesen ki lett kapcsolva ami alacsony képfrissítési ráta mellett kisebbre veszi az iterációk és gömbök számát.

A kezdeti pozícióhoz képest a kamera nem volt megmozdítva, valamint a tesztben érintett két paraméteren felül más nem lett átállítva a kezdeti alapértelmezettekhez

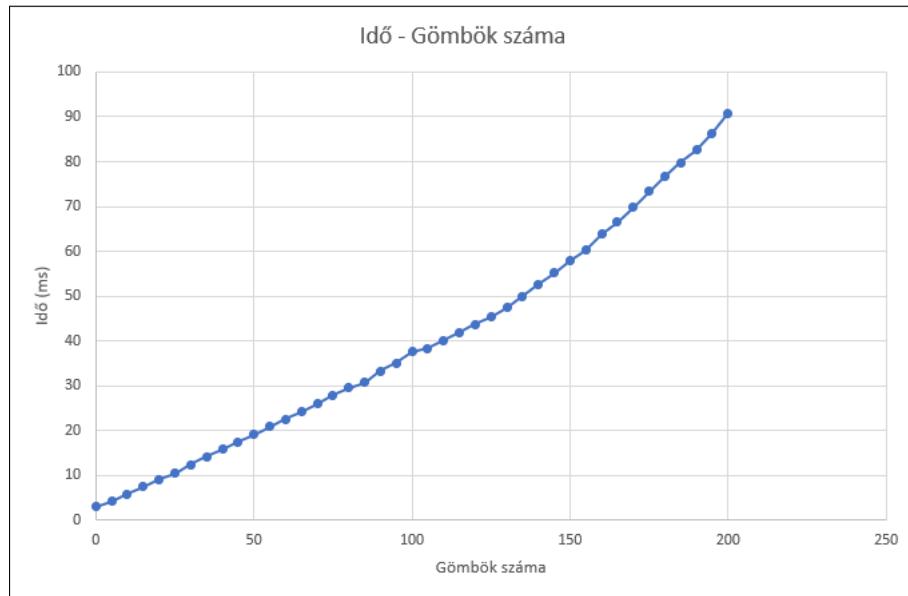
képest. A futás eredményéből készült grafikon a 3.12. ábrán látható.



3.12. ábra. Teljesítményteszt az iterációk számának függvényében

Megfigyelhető hogy az iterációk számától lineárisan függ a képfrissítési idő. Bármely két egymást követő sor különbsége 4-6 ms, illetve 10 iterációs eset ideje nagyjából fele a 20-nak és kb. harmada a 30-nak. Az is észrevehető hogy rettentően lelassítja az alkalmazást az iteráció növelése, elég 36-ig felmenni hogy a tesztelői környezeten 10 FPS alá essen a képfrissítési ráta (vagyis 100 ms fölé megy a frissítési idő), ami már határozattan nem folyamatos megjelenítést jelent.

Ha a 3.4. kód 15. sorát átírjuk **ballCount += 5** -re akkor azt is meg tudjuk vizsgálni hogy az mozgatható gömbök száma hogyan hat a képfrissítési időre. Ezen futás eredményéből készült grafikont a 3.13. ábra mutatja.

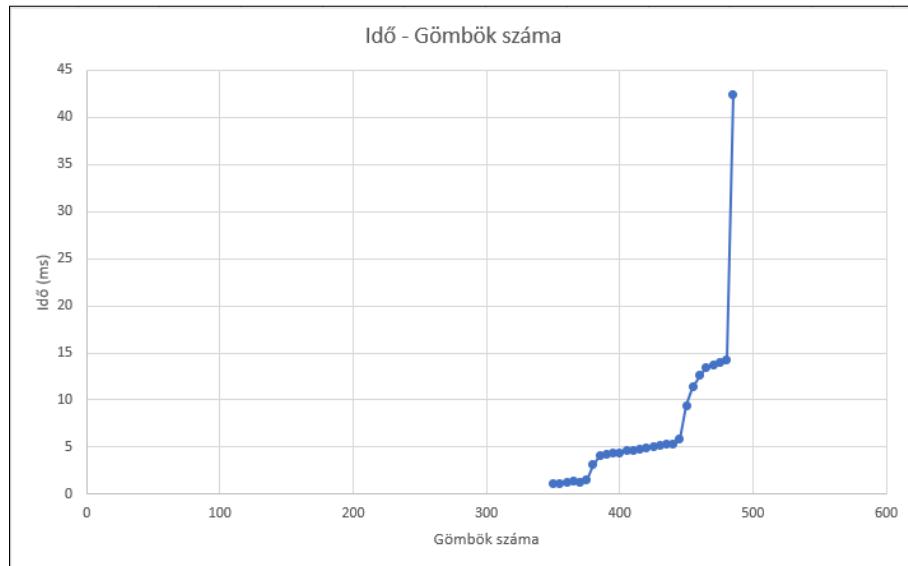


3.13. ábra. Teljesítményteszt a gömbök számának függvényében

Itt is nagyjából lineáris összefüggést tapasztalunk, két egymás utáni érték között 1-3 ms eltérés mutatkozik, illetve az is látható hogy jóval kevesebb hatása van a gömbök számának növelése a teljesítményre. A tesztkonfigurációnak 0 iteráció mellett 40 gömb még nem okoz problémát, ellenben 0 gömb mellett 40 iteráció az előző teszt tanulsága szerint már sokkal inkább.

A gömbök esetében is inkább a megjelenítés okozza a gondot, a `glDrawArrays()` sor ideiglenes kommentezésével a kirajzolást lényegében megszüntetjük, ám az ütközések modellezését nem befolyásoljuk. Ha ezután újra futtatjuk ez előbbi tesztet, akkor megtudhatjuk hogy a fizika kiszámolása mennyire lassította a kirajzolást. A futás eredményének egy részéből készült grafikon a 3.14. ábrán látható.

A grafikonra 350-nél kevesebb gömbhöz tartozó mérési értékek nem szerepelnek, az azokhoz tartozó számolási idő kevesebb mint 1 ms volt. Ez jól mutatja hogy a kirajzolási időhöz képest az ütközések kiszámolása jelentéktelen.



3.14. ábra. Teljesítményteszt a gömbök számának függvényében, kirajzolás nélkül

Az ábra alapján viszont itt már inkább exponenciális összefüggést állapíthatunk meg lineáris helyett. Az utolsó érték 490 gömbbel már 1000 ms volt, de ki lett hagyva az ábrázolás megkönnyítése érdekében. A teszt alatt azonban a magasabb értékek során a processzor összesített kihasználtsága a Windows Task Manager szerint 14%-os volt, és egyik szál sem mutatott állandó teljes terhelést.

A fizika futása nem lett több szálra bontva, így nem meglepő hogy csak **egyetlen processzormagot használ** igazán, azonban az furcsa hogy azt az egyet nem teljes mértékben. A jelenség pontos forrása ismeretlen, könnyen lehet hogy a hardver vagy az operációs rendszer korlátozza biztonsági okokból, ellenben a probléma csak extrém körülmények között jelentkezik, így nem lett sok idő fordítva az ok felkutatására.

A többi paraméter nincsen mérhető befolyással a teljesítményre. Fontos megjegyezni hogy a kirajzolás módja miatt a kamera pozíciója is befolyásolja a sebességet. Ha egyenesen felfelé nézünk, a sugár lépései jelentősen megnőnek, hiszen minden a legközelebb lévő felület távolságát lépjük előre, így néhány iteráció elegendő ahhoz hogy kiléphessünk a maximális távolsággal a **RayMarch()** ciklusából.

Ugyanezen okból kifolyólag a talajtól közel kiinduló, azzal párhuzamos sugarak kis lépésekben tudnak csak haladni, így több iteráció szükséges a **RayMarch()** ciklusából való kilépéshez, ami lassabb kirajzolási sebességet eredményez.

4. fejezet

Összegzés

Az elkészült program **teljesíti a kitűzött célokat**. A fraktálokkal való ütközés a kilőhető gömbök segítségével látványosan személhetetve van. A többi objektumnak hála az is látszik hogy **általános megvalósításról** van szó, azaz amit ki tudunk rajzolni azzal lényegében plusz befektetés nélkül automatikusan ütközni is tudunk.

A használt fraktálgenerálási módszer az iterációk szabad szabályozásával személetesen mutatja hogyan tevődik össze a fraktálunk. A fraktál dinamikus színezése és a véletlen generált fraktálok közötti átmenet **egyedi vizuális élményben részesíti** a felhasználót.

A kódban rengeteg fejlesztési potenciál van, melyek minden matematikai mind, programozási szempontból **érdekes kihívás elé állítanak**. Ezekről a következő fejezetben részletesebben tárgyalok.

A program megírása kitűnő lehetősőg a számítógépes grafikai ismeretének kiterjesztésére, nem is beszélve arról hogy a Sphere Tracing algoritmus **relatíve kevés kód** segítségével működésre bírható, és látványos eredményt tud produkálni, ennek köszönhetően a háttérismerekek elsajátítása után kifejezetten kellemes vele dolgozni.

A számítógépes grafika iránt érdeklődőknek minden képpen javaslom az általam is megvalósított alkalmazás megírását **tanulás, gyakolás vagy éppen szórakozás gyanánt!**

5. fejezet

További fejlesztési lehetőségek

A legnagyobb és legegyértelműbb fejlesztési lehetőség hogy az ütközés ne csak gömbökkel működjön, hanem **tetszőleges alakzattal**. Gömbökkel meglehetősen hatékonyan működik ez a módszer, ám ez csak annak köszönhető hogy a gömbökkel nagyon egyszerű meghatározni az ütközés pontját. Például egy kocka esetében tisztán távolságfüggvény segítségével ez már nem triviális, és ha a kocka orientációjával nem foglalkozunk az feltűnő, még egy mintázat nélküli gömböt nem kell forgatnunk hogy hihető mozgást biztosítsunk neki

Foglalkoztam a megvalósításával, de határidőre nem tudtam volna elkészülni vele, ezért inkább kihagytam. Némi kutakodás után nem találtam mást aki megvalósított volna tisztán távolságfüggvények segítségével általános, nem csak gömbökkel működő ütközés érzékelő algoritmust. Ami azt is jelentheti hogy ésszerű futási idő mellett nem is feltétlen lehetséges, így nem szégyenlem hogy ki kellett hagynom.

A gömbök egymással való ütközése jelenleg is kicsit furcsa, ami abból adódik hogy minden ütközés úgy van kiszámolva mintha egy mozdulatlan testtel történt volna. Ha figyelembe lenne véve hogy amivel ütközött az mozgásra képes objektum-e, illetve hogy annak milyen az aktuális sebessége akkor **valósághűbb lenne a labdák egymásnak ütközése**.

Nem kell nagy módosításokat tenni hogy a **GetDist()** függvény ne csak egy távolsággal hanem egy egyedi objektum azonosítóval is visszatérjen. Ha ez megvan akkor már csak egy megfelelő modellt kellene találni a labdák ütközésére. Például a sebességvektoraikat kicserélhetnénk kettejük között, úgy hogy mindkettő csökken valamelyest. Esetleg az egyik megkaphatná a saját és a másik vektor megfelelően

súlyozott átlagát és fordítva.

A gömbök **kilövése** lehetne úgyhogy egy (esetleg néhány) labda van középen amit minden kilövünk egyesével (vagy néhányasával), a többi pedig kering körülötte és kilövéskor a helyére ugranak. Ekkor a kilövést és a labdahívást külön billentyű szabályozná.

Az alkalmazás **teljesítményét** is lehetne javítani. Nagyon sok ismert mód van a Sphere Tracing algoritmus általános javítására, amiből én egyet sem alkalmaztam. Ezeket lehetne kombinálni a **felbontás** szabályozásával. Persze most is lehet szabályozni a felbontást az ablak átméretezésével, de elegánsabb lenne ha külön lehetne állítani a megjelenítési és a renderelési felbontást.

A fraktál generálási módján is sokat lehetne változtatni. minden iterációban 9 transzformációt végzünk el a fraktálunkon, ha néhány tengely menti transzformációt kihagynánk az sokat gyorsítana. Esetleg valami érdekesebb (nem alapvető) transzformáció segítségével kevesebb iteráció is elegendő lenne ugyanilyen részletes fraktál létrehozásához. Ennek megváltoztatása azonban részben filozófiai kérdés. Az eltolás, forgatás, tükrözés a legalapvetőbb geometriai transzformációk és alacsony iterációs szám nál még jól követhető hogy hogyan hatnak az alakzatra.

Ha a teljesítmény javult, a **fényszámításon** is sokat lehet javítani. Sphere Tracing algoritmussal akár közel fotorealisztikus fényeink is lehetnek. Persze találni kell egy kompromisszumot kinézet és gyorsaság között, de vannak jól ismert trükkök a fények szebbé tételeire amik csak minimális teljesítményromlást eredményeznek.

A felhasználói felület is kissé fapados, ennél szebb és hatékonyabb módja is lehetne az értékek bevitelére és leolvasására. Példának okáért egész kényelmesnek és gyorsnak hangzik, ha az értékeket úgy is lehetne szabályozni hogy billentyűlenyomással kiválasztjuk a szerkeszteni kívánt paramétert (mondjuk a számok 1-9-ig és a 0 lenne a nullázás) és egérgörgővel szerkesztjük.

Nem minden fraktál néz ki jól, vannak kifejezetten izgalmasan kinézők és vannak unalmasabbak. A pusztta értékek alapján nem egyértelmű hogy mi határozza meg ezt. Ezért jó lenne ha a szebb példányokat el lehetne menteni. Mivel csak 9 numerikus értékről lenne szó, így nem is lenne komplex a **mentés** és **betöltés**, de kellemes funkció lenne.

Irodalomjegyzék

- [1] *Raymarching Distance Fields: Concepts and Implementation in Unity.* <https://flafla2.github.io/2016/10/01/raymarching.html>. (Accessed on 05/10/2020).
- [2] *Sierpinski Triangle Fractal - The easiest - C++ Articles.* <https://jutge.org/doc/cplusplus.com/articles/LyTbqMoL/index.html>. (Accessed on 05/10/2020).
- [3] *Simple DirectMedia Layer - Wikipedia.* https://en.wikipedia.org/wiki/Simple_DirectMedia_Layer. (Accessed on 05/13/2020).
- [4] *OpenGL – Wikipédia.* <https://hu.wikipedia.org/wiki/OpenGL>. (Accessed on 05/12/2020).
- [5] *ocornut/imgui: Dear ImGui: Bloat-free Immediate Mode Graphical User interface for C++ with minimal dependencies.* <https://github.com/ocornut/imgui>. (Accessed on 05/12/2020).
- [6] *OpenGL Mathematics.* <https://glm.g-truc.net/0.9.4/api/index.html>. (Accessed on 05/13/2020).
- [7] *File:Ray trace diagram.svg - Wikimedia Commons.* https://commons.wikimedia.org/wiki/File:Ray_trace_diagram.svg. (Accessed on 05/13/2020).
- [8] *iq - Shadertoy BETA.* <https://www.shadertoy.com/user/iq>. (Accessed on 05/23/2020).
- [9] *Grafika BSc Gyakorlat anyagok – ELTE Számítógépes Grafika.* <http://cg.elte.hu/index.php/grafika-bsc-gyakorlat-anyagok/>. (Accessed on 05/13/2020).

- [10] *1.3: Dot Product - Mathematics LibreTexts.* [https://math.libretexts.org/Bookshelves/Calculus/Book%3A_Vector_Calculus_\(Corral\)/01%3A_Vectors_in_Euclidean_Space/1.03%3A_Dot_Product](https://math.libretexts.org/Bookshelves/Calculus/Book%3A_Vector_Calculus_(Corral)/01%3A_Vectors_in_Euclidean_Space/1.03%3A_Dot_Product). (Accessed on 05/14/2020).
- [11] *Python & OpenGL for Scientific Visualization.* <https://www.labri.fr/perso/nrougier/python-opengl/>. (Accessed on 05/24/2020).
- [12] *Spherical coordinate system - Wikipedia.* https://en.wikipedia.org/wiki/Spherical_coordinate_system#Coordinate_system_conversions. (Accessed on 05/26/2020).

Ábrák jegyzéke

1.1.	Sphere tracing: A kamerából kiinduló fénysugár mindenkorának annyit halad előre amekkora a hozzá legközelebb lévő felület távolsága [1]	2
1.2.	Példa egy IFS-re: a Sierpiński háromszög néhány iterációja [2]	3
2.1.	Fő programablak	5
2.2.	Terminálablak	5
2.3.	A „Parameters” feliratú panel felső harmada	6
2.4.	A "Parameters" feliratú panel középső harmada	6
2.5.	A "Parameters" feliratú panel alsó harmada	8
3.1.	Az SDL logója	11
3.2.	Az OpenGL logója	11
3.3.	A GLM logója	12
3.4.	A raycast technika ábrázolása. [7]	12
3.5.	Sphere tracing: A kamerából kiinduló fénysugár mindenkorának annyit halad előre amekkora a hozzá legközelebb lévő felület távolsága [1]	13
3.6.	A fénymodell különböző komponensei.	14
3.7.	A CMyApp osztálydiagramja (két részre szedve a hosszúsága miatt)	15
3.8.	A skaláris szorzat előjele és a vektorok közötti szög összefüggése [10]	17
3.9.	NDC (Normalized Device Coordinates) - az ablak koordinátáit normalizáltuk, hogy a ball alsó sarok (-1,-1), a jobb felső sarok (+1,+1) legyen. [11]	19
3.10.	A gCamera osztálydiagramma	20
3.11.	Polár koordináta [12]	22
3.12.	Teljesítményteszt az iterációk számának függvényében	26
3.13.	Teljesítményteszt a gömbök számának függvényében	27
3.14.	Teljesítményteszt a gömbök számának függvényében, kirajzolás nélkül	28

Forráskódjegyzék

3.1.	A sphere tarcing algoritmust megvalósító kód	13
3.2.	A felületi normálist kiszámoló függvény	17
3.3.	Az eye és at vektorok frissítése	20
3.4.	A tesztelést végző kód	25