



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	11
3. Fejlesztői dokumentáció	12
3.1. Fejlesztői eszközök	12
3.2. Képalkotási módszer	14
3.2.1. Fénymodell	16
3.2.2. Távolságfüggvények	17
3.3. Megvalósítás	19
3.3.1. CMyApp osztály	19
3.3.2. gCamera osztály	23
3.3.3. Fragment shader	26
3.4. Tesztelés	29
3.4.1. Működés helyessége	30
3.4.2. Teljesítmény	32
4. Összegzés	36
5. További fejlesztési lehetőségek	37
Irodalomjegyzék	39
Ábrajegyzék	41
Forráskódjegyzék	43

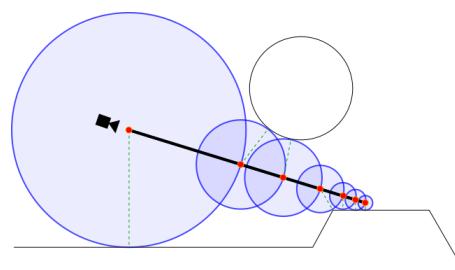
1. fejezet

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 akár pattanjanak le a másikról. Ezen viselkedés kiszámolása meglehetősen költséges tud lenni, ráadásul különböző alakzatoknál különböző algoritmusokat kell használni.

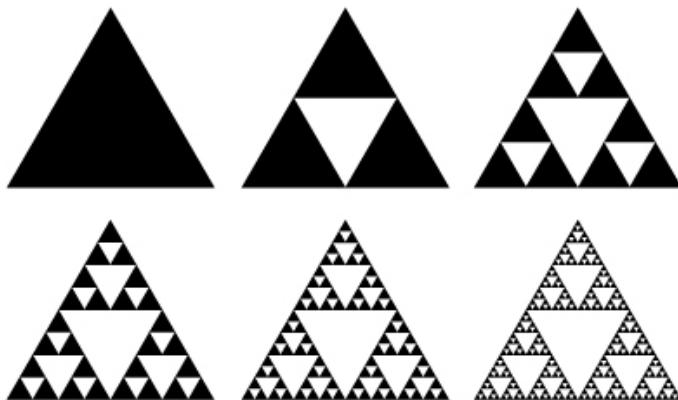
Szakdolgozatomban azzal foglalkozom hogy hogyan lehet a kirajzolást és az előbb említett valósághű viselkedést fraktálokkal elvégezni. A velük való ütközéshez megállapításához ugyanaz fog segítséget nyújtani, mint a kirajzolásukhoz: távolságfüggvények.



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 algoritmus van.



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 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

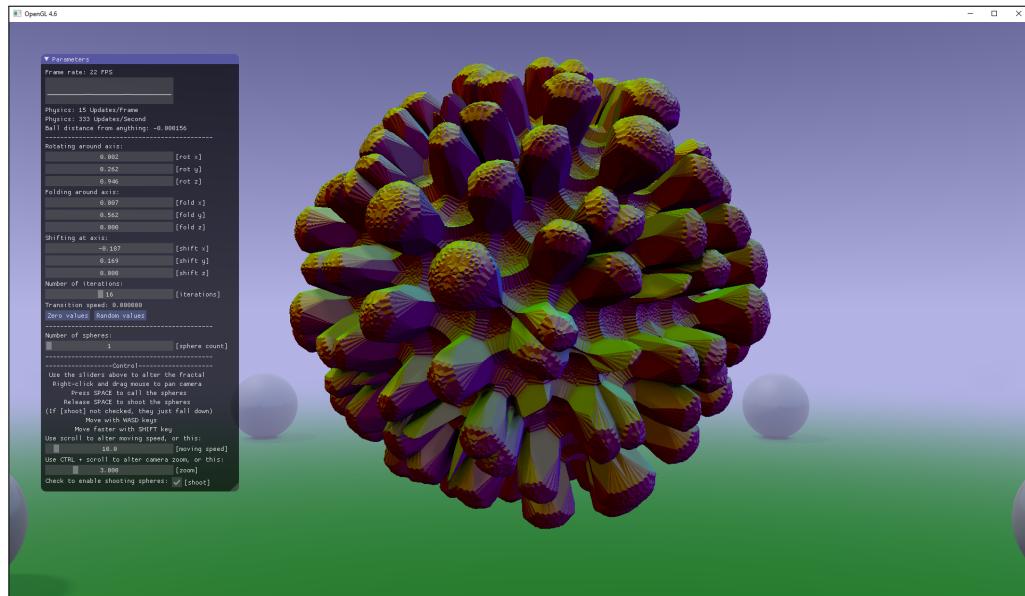
Felhasználói dokumentáció

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

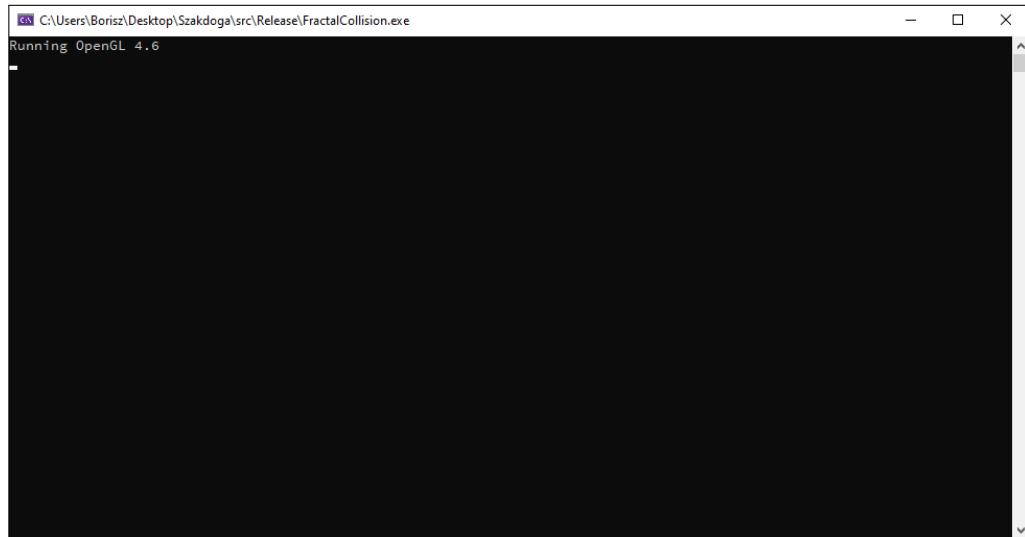
A program sikeres indítása után – melyről a 2.2. fejezetben tudhatunk meg többet – kettő darab ablakkal találjuk szembe magunkat. Ezekről a 2.1. és 2.2. ábrák mutatnak egy-egy képernyőképet.

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 a futás különböző paramétereit tudjuk nyomon követni és módosítani. Az ablak alapértelmezetten 1720x900 nagyságú, de szabadon átméretezhető, viszont az ablak mérete befolyással van a teljesítményre! Mindig az ablak pontos felbontásában fog renderálni, 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 rendelkezik.

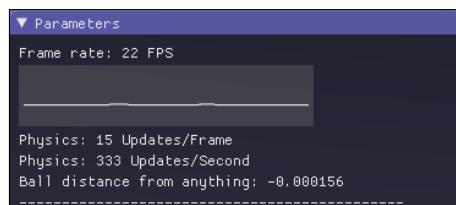


2.2. ábra. Terminálablak

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 2.7. 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.

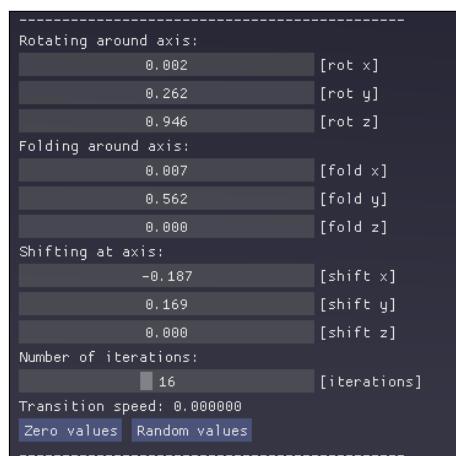


2.3. ábra. A „Parameters” feliratú panel felső harmada

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

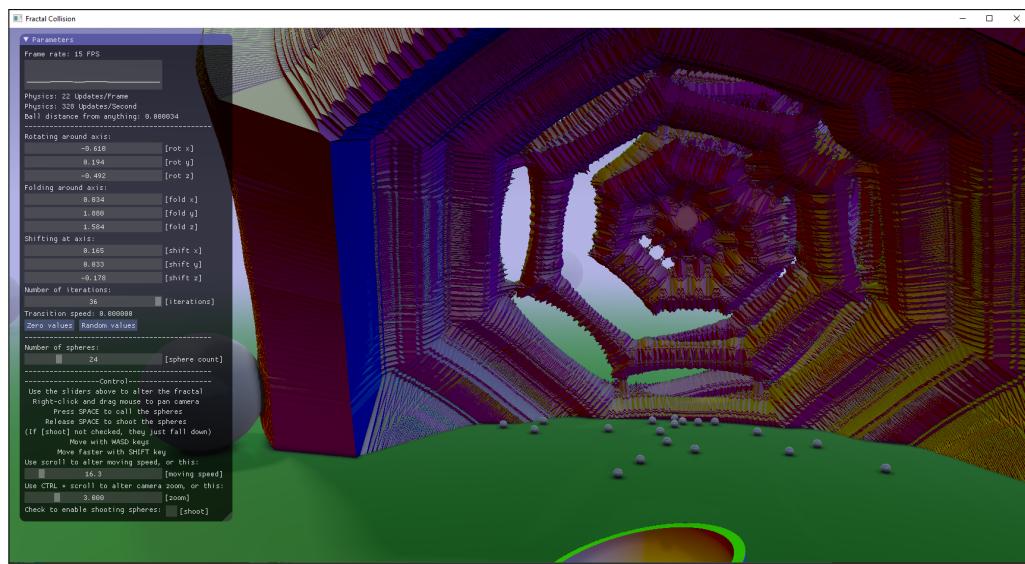
A két „**Physics:**” felirat után olvashatjuk le hogy milyen gyakran van a labdák mozgása frissítve frissítés/képkocka és frissítés/másodperc mértékegységekben. Egy frissítés során minden labda sebessége és pozíciója újraszámolódik, valamint ellenőrzésre kerül az is hogy ütközött-e valamivel.

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 hagyományos csúszka, csak nincsen minimum és maximum értéke és az egeret tovább is lehet húzni mint a csúszka vége. Mindegyik csúszkának CTRL + kattintással, vagy duplakattintással begépelezett értéket is meg lehet adni.

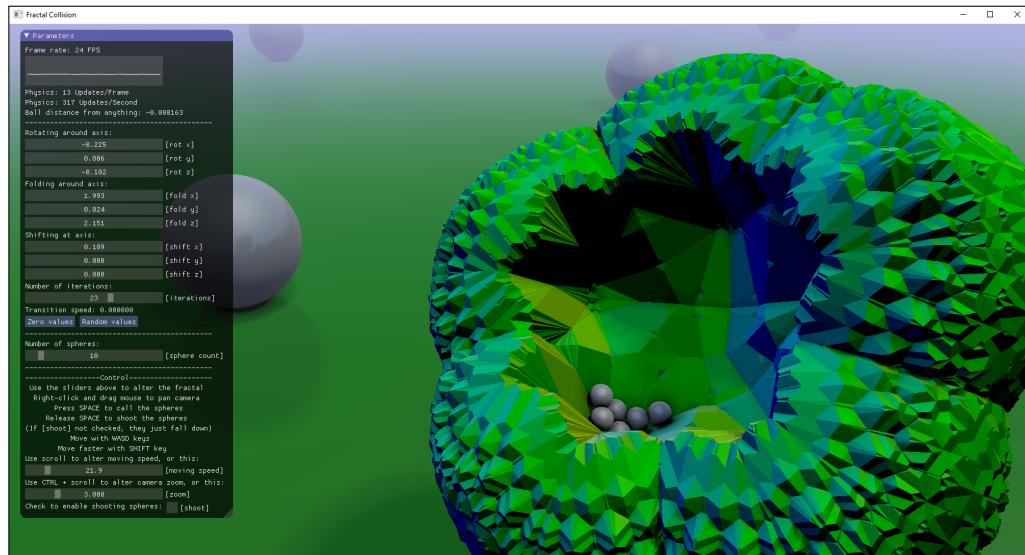


2.5. ábra. A megfelelő paraméterekkel látványos fraktálokat kaphatunk

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 tükrözésík ami a megadott szöggel (radián) fordul el és tükrözi az a sík egyik oldalán lévő pontokat a másikra – de csak az egyik oldalt, így ez nem egybevágósági transzformáció, ezért nem tükrözésként van elnevezve. Itt a három érték 3 különböző tükrözésíket 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.



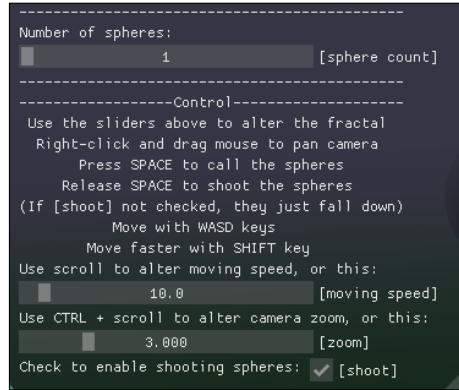
2.6. ábra. Bármilyen paraméter esetén képesek a labdák helyes ütközésre

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, csak a többi fraktálra vonatkozó paramétere van hatással.

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

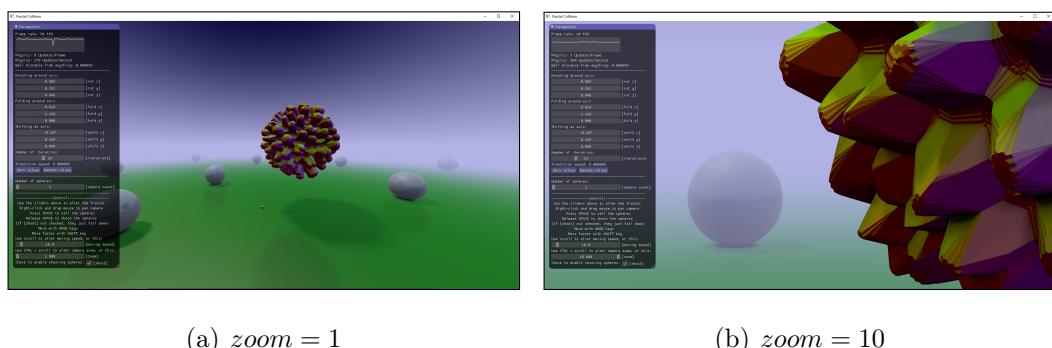


2.7. ábra. A „Parameters” feliratú panel alsó harmada

Az alsó harmadban (2.7. 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.

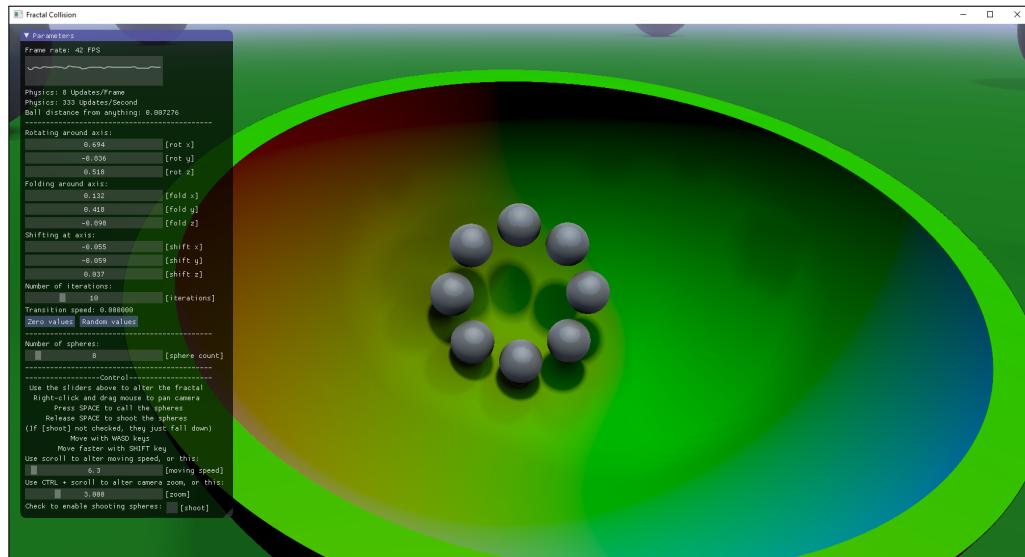
Van egy rövid ismertető szövege a panelnek, ami azt a célt szolgálja hogy ezen dokumentáció nélkül is tudja használni egy felhasználó ha leül a program elé. Ebben kerülnek ismertetésre a virtuális tér bezárásához szükséges irányítások is. A mozgásra a **WASD** billentyűket és az egeret lenyomott jobb 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 megnégyezzéződik a sebesség, felengedésére visszaáll. A kamera látószögét lehet csökkenteni a **[zoom]** csúszka értékének növelésével, vagy a **CTRL + görgő** segítségével is.



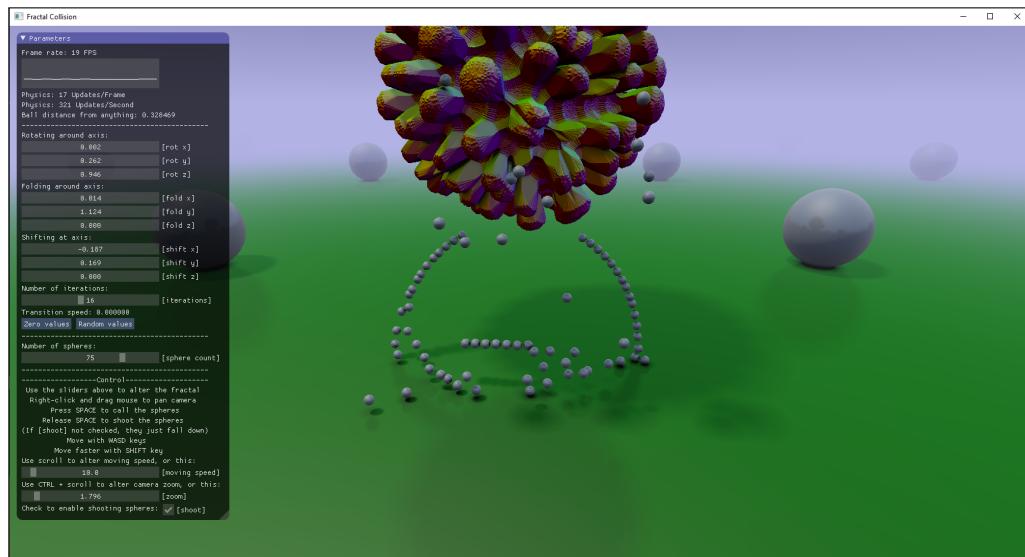
2.8. ábra. A **[zoom]** paraméter hatása a képre (a kamera mozgatása nélkül)

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



2.9. ábra. A labdák helyzete a szóköz lenyomása közben

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 körön belül mentén egyenletesen elhelyezkedve, mely a 2.10 ábárn is megfigyelhető. A szóköz billentyűt felengedve egyszerre kilövődnek a labdák, ha be van pipálva a panel alján a **[shoot]** jelölőnégyszet, ha nincsen bepipálva csak leesnek a gravitációnak megfelelően.



2.10. ábra. A labdák kilövése utáni pillanatkép

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 nincsenek 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 sarokban az ablak bezárás gombját használni. Ha bezárjuk a terminálablakot akkor minden ablak bezárul, ha először a fő programablakot zárjuk be akkor utána előfordulhat hogy még külön be kell zárni a terminálablakot – az esetek többségében azonban magától bezárul.

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** kiterjesztésű 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)

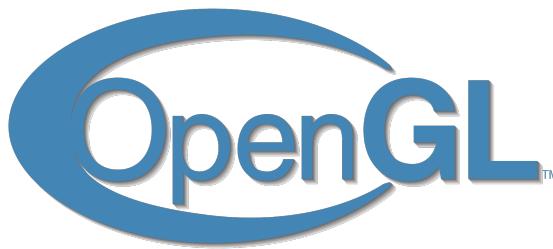
A Simple DirectMedia Layer (SDL) könyvtár egy crossplatform multimédiás könyvtár, ami alacsony szintű, hatékony hozzáférést ad audio, bemeneti (egér, billentyűzet, joystick), valamint grafikus (OpenGL-en keresztül GPU-hoz) eszközökhöz. Az alkalmazásban az SDL 2.0 van használva. [3]



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 könyvtárra csak néhány szimpla forrásfájl beillesztésére. [5]

GLM

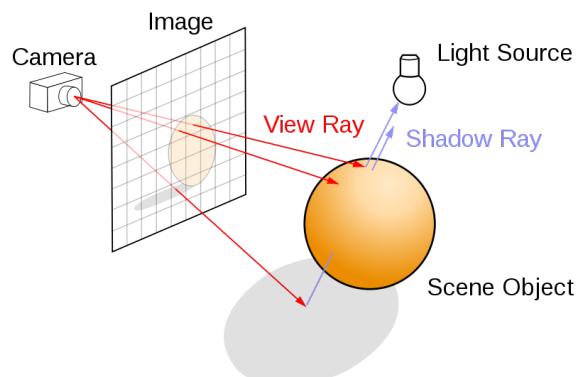
Az OpenGL Mathematics egy headerben importálható C++ könyvtár mely a GLSL specifikációira szabva (elnevezések, funkciók) tartalmaz számtalan matematikai függvényt. Segítségével a GLSL-ben használatos függvények és típusok C++-ban is használhatóak. [6]



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]

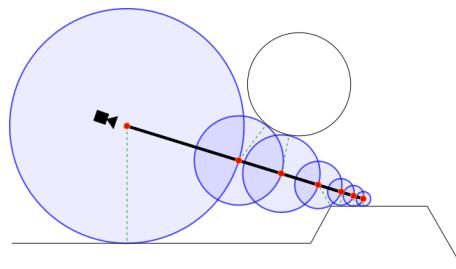
A **sphere tracing** algoritmus egy speciális esete az úgynevezett **raymarching** algoritmus-családnak. Működéséhez a kirajzolni kívánt objektumokat távolságfüggvényekkel kell reprezentálni, mely a virtuális tér bármely pontjáról képes meghatározni hogy milyen messze van az objektum felületétől. Az 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 (**ro**) és egy vektort (**rd**), melyek meghatározzák a sugár kezdőpontját és irányát. Ezen irány mentén lépked folyamatosan minden annyit, amennyi 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 minden csak annyit halad előre amekkora a hozzá legközelebb lévő felület távolsága [1]

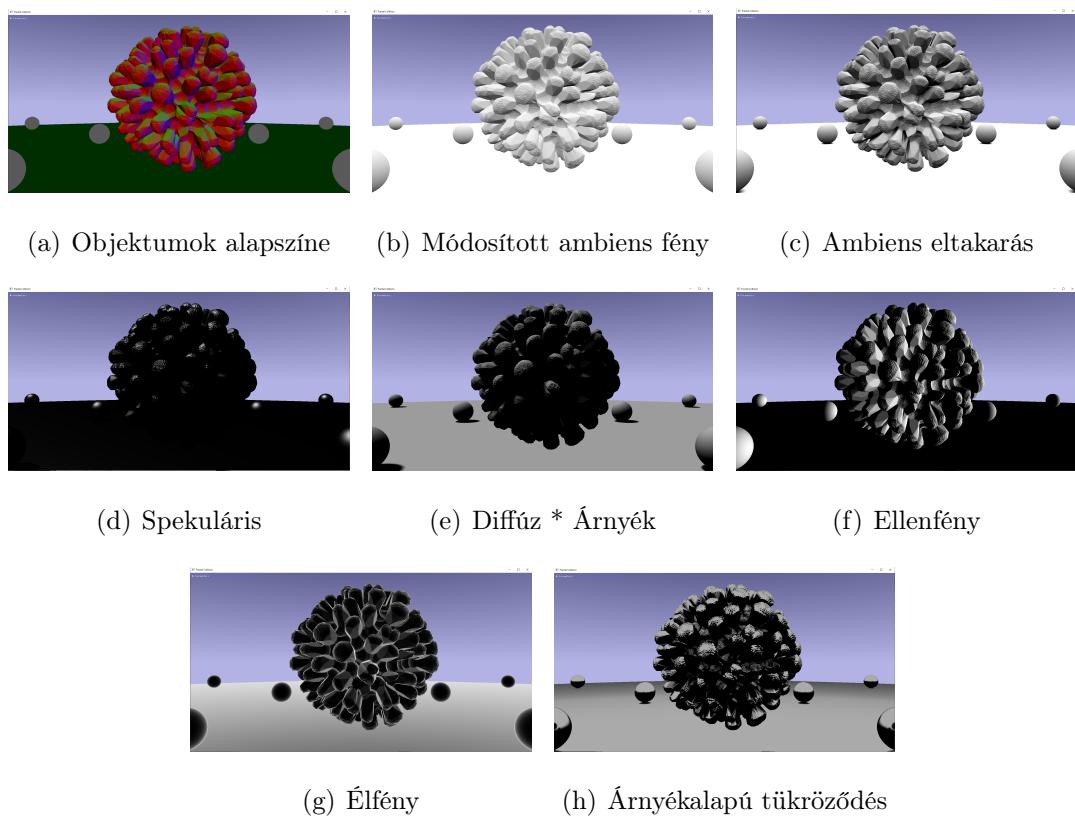
Az alkalmazásban teljesítményjavítás érdekében másik feltétel ($dS < 0.01 * dist / MAX_DIST$) lett használva a felületi távolsághoz, mely számításba veszi hogy mennyire messzire vagyunk a kamerától. Ha messzebb vagyunk

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 a végső színáranyalatot, melyre még egy gamma korrekció és egy ködszerű hatás is alkalmazva lett – ez utóbbi arra szolgál hogy a viszonylag kicsi maximális távolságot leplezze.



3.6. ábra. A fénymodell különböző komponensei.

3.2.2. Távolságfüggvények

Távolságfüggvényekkel reprezentáljuk az objektumokat. Ebből adódóan nagyon fontosak a megjelenítés szempontjából. Ebben az alfejezetben matematikai szempontból kerülnek részletesebb ismertetésre. A távolságfüggvények matematikai tár-gyalását Bálint Csaba munkája [9] alapján részletezzük.

1. Definíció. Legyen az (\mathbb{R}^3, d) metrikus térben egy $x \in \mathbb{R}^3$ pont és egy $A \subset \mathbb{R}^3$ halmaz távolsága az alábbi:

$$d(x, A) := \inf_{a \in A} d(x, a)$$

2. Definíció. Az $f : \mathbb{R}^3 \rightarrow \mathbb{R}$ akkor távolságfüggvény, ha

$$f(\mathbf{p}) = d(\mathbf{p}, \{f = 0\}) \quad (\mathbf{p} \in \mathbb{R}^3)$$

Vagyis a távolságfüggvény az ábrázolni kívánt alakzat felületén 0 értéket vesz fel, mindenhol máshol pedig ettől a felülettől vett távolságot. Ennek egy kiterjesztése az **előjeles távolságfüggvény**, mely a távolságnak negatív előjelet ad az általa reprezentált alakzaton belül. Az alkamazásban ilyen távolságfüggvények kerültek felhasználásra.

3. Definíció. Az $f : \mathbb{R}^3 \rightarrow \mathbb{R}$ pontosan akkor előjeles távolságfüggvény, ha létezik $D \subset \mathbb{R}^3$ halmaz, melyre

$$f(\mathbf{p}) = \begin{cases} d(\mathbf{p}, \text{bound}(D)) & \text{ha } \mathbf{p} \in D \\ -d(\mathbf{p}, \text{bound}(D)) & \text{ha } \mathbf{p} \notin D \end{cases}$$

ahol $\text{bound}(D) = \bar{D} \setminus \text{int}(D)$ a halmaz határa.

Nagyon sok matematikai alakzatnak ismert az előjeles távolságfüggvénye. Az alkalmazásban csupán néhány egyszerűbb alakzat lett felhasználva:

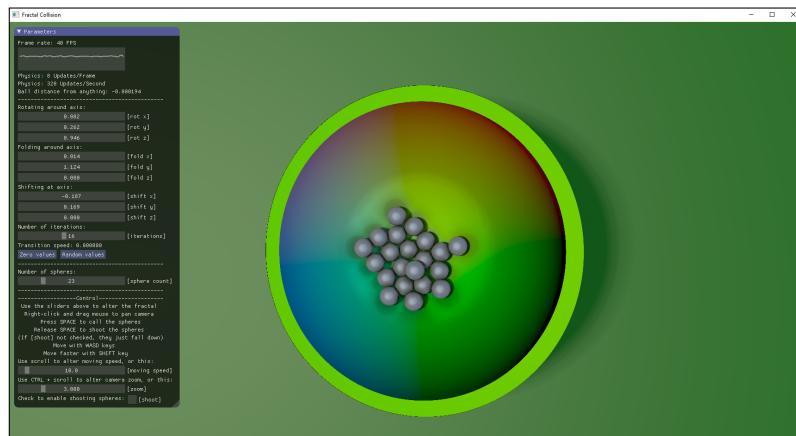
Felület	Paraméterei	Előjeles távolságfüggvények
Sík	$n \in \mathbb{R}^3$ normális $\ n\ _2 = 1$	$f_n(\mathbf{p}) = \langle \mathbf{p}, n \rangle$
Gömb	$0 < r \in \mathbb{R}$ sugár	$f_r(\mathbf{p}) = \ \mathbf{p}\ _2 - r$
Téglatest	$a, b, c \in \mathbb{R}$ oldalhosszak	$\mathbf{d} := [d_x, d_y, d_z]^\top := [x - a, y - b, z - c]^\top,$ akkor $f_{a,b,c}(x, y, z) =$ $\min \{\max \{d_x, d_y, d_z\}, 0\} + \ \max\{\mathbf{d}, \mathbf{0}\}\ _2$

Az előjeles távolságfüggvénnnyel leírt alakzatokkal könnyen végezhetünk műveleteket. Legyen $f, g \in \mathbb{R}^3 \rightarrow \mathbb{R}$ előjeles távolságfüggvények. Ekkor megkapjuk az általuk leírt felületek

- a) **unióját**, ha vesszük f és g minimumát
- b) **metszetét**, ha vesszük f és g maximumát
- c) **különbségét**, ha vesszük f és $-g$ (vagy $-f$ és g) maximumát

Az így kapott függvények már csak előjeles távolságfüggvény-becslések lesznek. Ha egyszerre több alakzatot is ki akarunk rajzolni akkor a függvények unióját kell venni. Egy ilyen unió eredményét adja vissza a Sphere Tracing algoritmusban használt **GetDist()** eljárás.

Továbbá ezen műveletekkel akár új alakzatot is alkothatunk! Például két koncentrikus gömb különbségéből ha kivonunk egy síkot akkor egy tál szerű alakzatot kaphatunk (3.7. ábra), melyben kitűnően lehet tesztelni a labdák egymásnak ütközését.



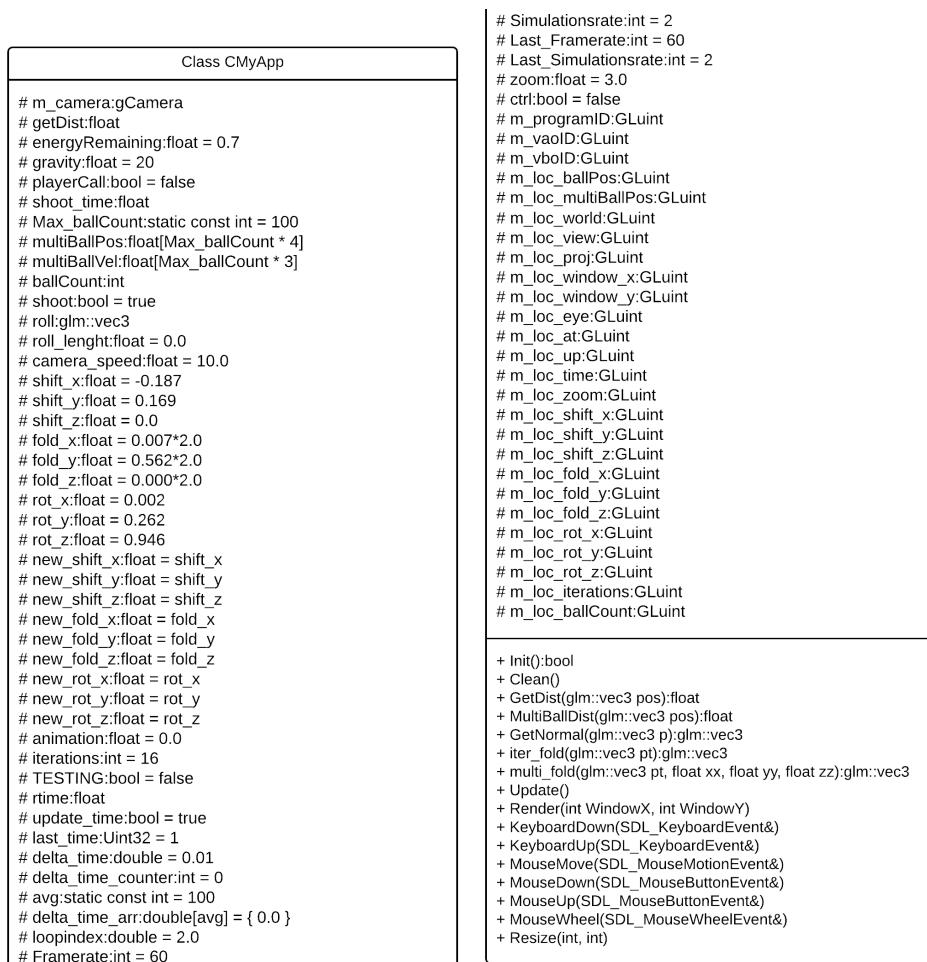
3.7. ábra. A kiindulási pozíció alatt található alakzat néhány labdával

3.3. Megvalósítás

A program írása során a Számítógépes Grafika BSc gyakorlat tárgy honlapjának [10] 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 bemeneteit, szimulálja a fizikát és továbbítja a shaderbe a paramétereket hogy megtörjenjen a kirajzolás. A fontosabb függvények külön részletezésre kerülnek.



(a) A diagram felső fele

(b) A diagram alsó fele

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

bool Init()

Ebben inicializálódik minden aminek kell, itt foglaljuk le a grafikus erőforrásokat. Definiálunk két darab háromszöget melyek összerakva egy olyan négyzetlapot alkotnak ami az XY síkot (-1,-1)-től (1,1)-ig lefedik. Ez a négyzetlap lefedi a teljes ablakot és ennek a fragmanet shaderben való átszínezésével kapjuk meg a képet. Az uniform változók memóriacímeit és a shadereket 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 **az ütközések ellenőrzése** és a labdák mozgatása a pozícióik frissítése által.

A labdák befogott pozíciói – amikhez akkor közelítenek ha lenyomjuk a szóköz billentyűt – itt kerülnek kiszámításra 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 darabszámától 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 a labdák az imént kiszámolt pozíciójának és a jelenlegi pozíciójának különbségének konstans-szorosát kapja meg sebességül. 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 hogy valamivel ütköztünk.

A helyes **viselkedés megállapításához** ismerünk kell az ütközés pontjában a felületi normálist. A normális közelítő értékének kiszámolásához használt módszer

(3.2. kódrészlet) 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.

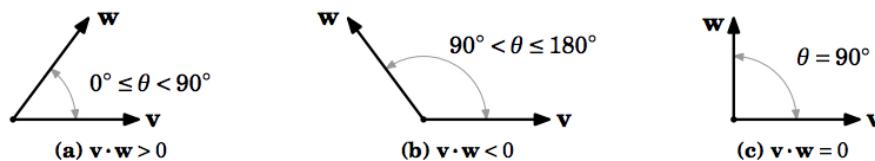
```

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 normális által meghatározott síkra kell visszavernünk³ a vizsgált labdának sebességvektorát és a megfelelő komponenseit némi leg csökkenetben, 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 – a mozgási energia egy része ugyanis perdületivé alakul.

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 belevisszük 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.



3.9. ábra. A skaláris szorzat előjele és a vektorok közötti szög összefüggése [11]

³Ez a normálisra való tengelyes tükrözést és egy -1 -el való szorzást jelent

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 sebességektor, 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. (3.9. ábra)

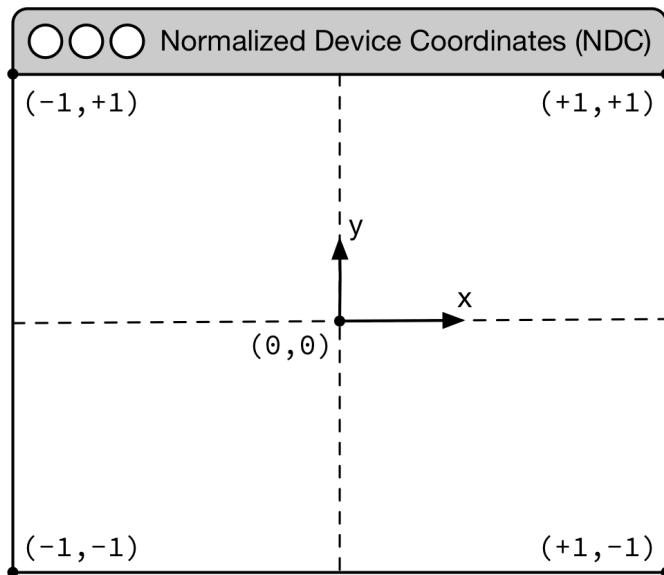
void Render(int WindowX, int WindowY)

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 transzfomáció 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.

A megadott négyzetünk a várt koordináták miatt (3.10. ábra) lefedi a teljes ablakot, viszont ha az ablak nem négyzet alakú, akkor torzítás történik. A vertex shaderben az ablak méreteinek ismeretében kompenzáljuk a torzítást, azáltal hogy a továbbadott vektor x komponensét megszorozzuk az ablak szélességének és magasságának hányadosával.



3.10. ábra. NDC (Normalized Device Coordinates) - az ablak koordinátái [12]

Ennek köszönhetően a fragment shaderben a bemeneti vektorunk már torzításmentesen reprezentálja az ablakunk koordinátáit. Ezután a fragment shaderben a 3.2. fejezetben ismertetett elmelet alapján megalkotára kerül a kép.

3.3.2. gCamera osztály

Ez az osztály egy általános megvalósítása egy virtuális kamerának és közeli-az egyben lett felhasználva a Számítógépes Grafika BSc gyakorlat tárgy honlapjának [10] 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.



3.11. 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** pozícióvektorainkhöz (**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 értéke hamis, akkor az **m_speed** változó értékét növeljük a négyeszeresére, illetve az **m_slow-t** igazra állítjuk. Azért szükséges ezen változó figyelése mert csak egyszer szeretnénk meg-négyezni a sebességet.

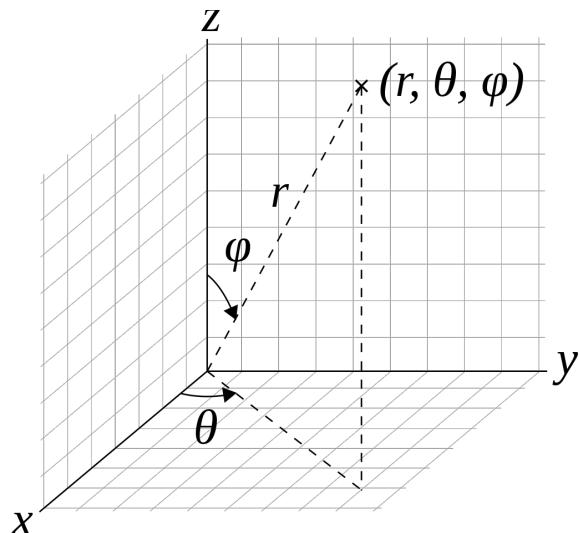
void KeyboardUp(SDL_KeyboardEvent& key)

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.



3.12. ábra. Polár koordinák [13]

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.

3.3.3. Fragment shader

A megjelenítés lényegi munkáját a fragment shader végzi, a 3.2. alfejezetben a képalkotás elméleti része már nagy vonalakban ismertetve lett, ebben az alfejezetben ennek az implementációját mutatom be. A függvények ismertetésének sorrendje nagyságrendileg a meghívásuk sorrendjét követi.

void main()

Az unifom változóként megkapott **eye**, **at** és **up** vektorok és a vertex shaderből érkező normalizált ablakkoordináták segítségével, valamint néhány vektorművelet felhasználásval kiszámoljuk a sugarak kiinduló pontját és irányát. Ezt átadjuk a **render()** függvénynek ami visszatér az adott pixel színével.

vec3 render(vec3 ro, vec3 rd)

A paramétereivel meghívja a **RayMarch()** függvényt, mellyel meg tudjuk határozni hogy az aktuális sugár a virtuális terünk mely pontjában ütközött. Ebben a pontban a **GetNormal()** függvénnnyel meghatározzuk a felületi normálist. Ezek ismeretében az alkalmazott fénymodellünkkel⁴ meg tudjuk állapítani hogy milyen színű legyen a paraméterként megkapott sugárnyalábhhoz tartozó pixelünk és ennek az rgb kódjával tér vissza a függvény.

vec3 GetNormal(vec3 p)

A paraméterként kapott pontban kiszámolja a felületi normális egy közelítését. Ezt meghatározzuk a differenciaképlet (3.1. képlet) alapján az adott pontban és az eredményt normálizáljuk. Az ϵ értékét az implementációban 0.001-nek választottam. Ha az érték túl nagy, akkor pontatlan lesz a közelítés, ha túl kicsi akkor pedig a számábrázolás pontatlansága okozhatja a kapott nomra pontatlanságát.

$$f'(x, y, z) \approx \frac{1}{\epsilon} \cdot \begin{bmatrix} f(x, y, z) - f(x - \epsilon, y, z) \\ f(x, y, z) - f(x, y - \epsilon, z) \\ f(x, y, z) - f(x, y, z - \epsilon) \end{bmatrix} \quad (3.1)$$

float RayMarch(vec3 ro, vec3 rd)

A 3.2. fejezetben az algoritmus implementációja és leírása is érintve lett. Meghívódik benne a **GetDist()** névvel illetetett távolságfüggvény. A felületi távolsággal kapcsolatos kilépési feltétel javításán kívül ez egy szabvány implementációja a **Sphere tracing** algoritmusnak.

⁴A 3.2.1. fejezetben a fénymodellről már volt szó, az implementációját nagyon hosszan lehetne részletezni de többnyire szabány megoldások és önmagyarázó kódokkal készült, így erre nem kerül sor.

float GetDist(vec3 pos, out float col)

A paraméterül kapott **pos** ponttól kiszámolja az összes felület uniójától vett távolságot. A testek felületét leíró függvények Ingio Quilez honlapján [14] található gyűjteményből származnak. Összességében a sík, a téglatest és a gömb függvényei vannak különféle módon tarszformálva. minden alakzathoz tarozó függvény visszatér a megfelelő távolsággal, majd ezeknek a minimuma lesz a függvény visszatérési értéke.

A függvénynek van egy **out** paramétere is, amit nem kell feltétlenül megadnunk. Ez a paraméter annak függvényében kap értéket hogy melyik objektumtól származik a legkisebb távolságérték. Ez az érték akkor kerül ellenőrzésre a **render()**-ben amikor a **RayMarch()** segítségével már megállípatottuk hogy hol állt meg a kiküldött sugárnyaláb. Ezen érték felhasználásával minden objektumnak adhatunk egy a fényektől és megvilágításától független alapszínt.

Ez az alapszín egyébként a fraktál esetén a felületi normális átkonvertálva **rgb** értékekbe. Ugyanezt a színezési technikát kapja a a kezdeti pozíció alatt található félgömbhéj is.

A fraktálunk pedig egy egyszerű téglatest amire az **iter_fold()** transzofmációs függvény lett alkalmazva.

vec3 iter_fold(vec3 pt)

Paraméterül egy pontot kap, amire az adott traszformációkat elvégzi, majd a transzformált ponttal tér vissza. Ez a függvény egy ciklust tartalmaz, mely az unifom változóként megkapott **iterations**-szor fogja megismételni minden tengelyre az eltolás, forgatás és fold traszformációkat – amiket unifom változóként megkapott paraméterekkel hajt végre – egymás után a kapott pontra.

Az eltolás pont esetében csak annyit tesz hogy a megfelelő koordinákból kivonjuk az eltolás koordinátáit.

A forgatás a klasszikus 3 dimenziós forgatási mátrixokra épül:

$$R_{x,\vartheta} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos(\vartheta) & -\sin(\vartheta) \\ 0 & \sin(\vartheta) & \cos(\vartheta) \end{bmatrix}$$

$$R_{y,\vartheta} = \begin{bmatrix} \cos(\vartheta) & 0 & \sin(\vartheta) \\ 0 & 1 & 0 \\ -\sin(\vartheta) & 0 & \cos(\vartheta) \end{bmatrix}$$

$$R_{z,\vartheta} = \begin{bmatrix} \cos(\vartheta) & -\sin(\vartheta) & 0 \\ \sin(\vartheta) & \cos(\vartheta) & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Fold (kvázi tükrözés) egy eltolás a megadott síkra merőlegesen, úgy hogy annyival kerül eltolásra amilyen távolságra van a megadott síktól. Csak a sík egyik oldalán lévő pontokkal végezzük el ezt, a másik oldalán lévő pontokat nem mozdítjuk. Ezáltal már nem lesz egybevágósági traszformáció, és emiatt nem is mondható tükrözésnek, ezért lett a **fold** kifejezés használva rá.

Fontos hogy a transzformációk között legyen olyan ami nem egybevágósági transzformáció, amennyiben nem lenne akkor hiába iterálnánk akármennyit, az alakzat formája nem változna. Lényegében ez az egyetlen kikötése a fraktál generátoroknak, amíg ez teljesül addig tetszőlegesen variálhatjuk a traszformációkat hogy hatékonyabban lehessen kiszámolni az alakzatot, vagy éppen izgalmasabb formákat kapunk.

3.4. Tesztelés

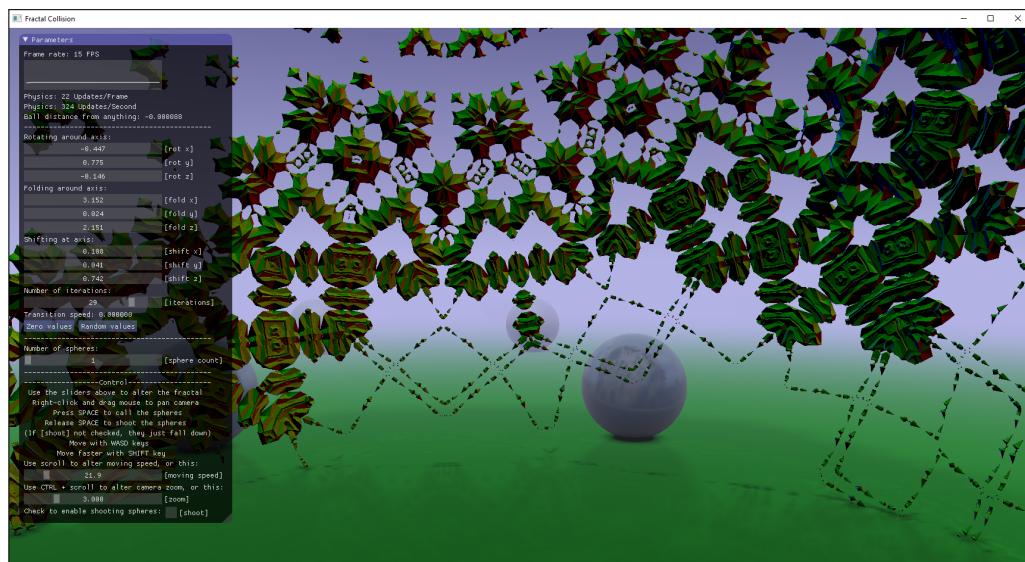
A tesztelés során megvizsgáltam hogy a funkciók (2.1. fejezet) az elvártaknak megfelelően működne-e. Az ehhez használt számítógép konfiguráció:

- 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:

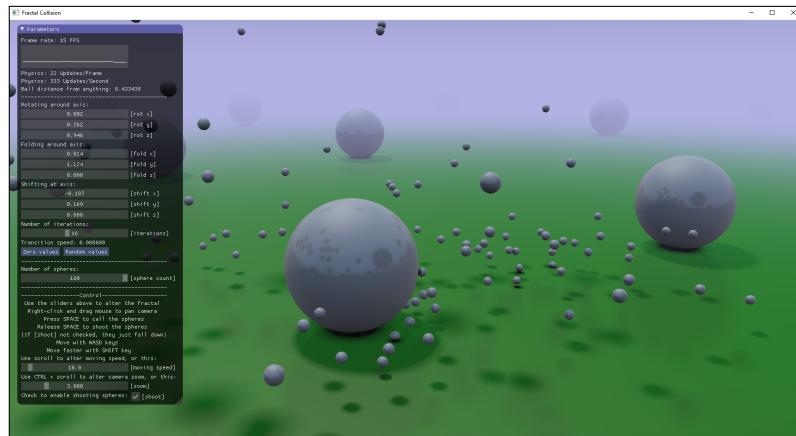
1. A virtuális kamerát lehet forgatni az **egérrel**;
2. A virtuális kamera nagyítását lehet állítani a **CTRL + görgővel** és a csúszkával is;
3. A virtuális kamerát lehet mozgatni a **WASD** billentyűkkel;
4. A mozgás sebességét lehet gyorsítani a **SHIFT** billentyűvel és állítani a görögővel vagy 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;



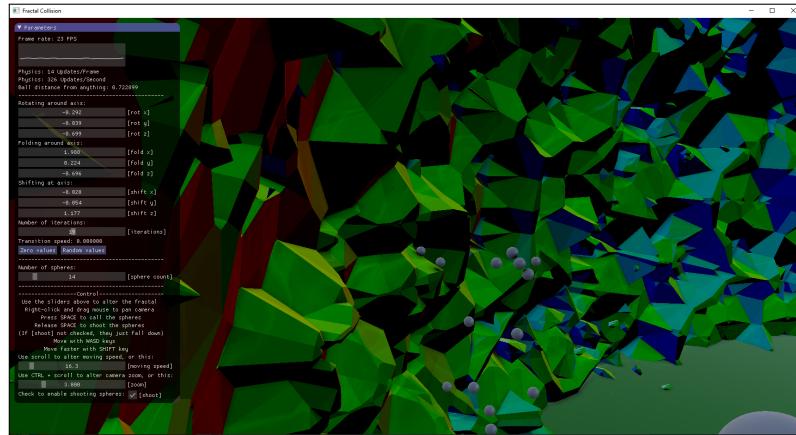
3.13. ábra. A paraméterek egy lehetséges kombinációja amikor már éppen csak látható a fraktál, kicsit módosítva a megfelelőkön már nem látható

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 (3.13. ábra), minél nagyobb az iteráció értéke annál gyakrabban;

10. A **SPACE** billentyű nyomva tartásakor a labdák megjelennek a felhasználó előtt. Ha közel a akadályba ütköztek akkor nem tudnak;
11. A **SPACE** billentyű felengedésekor a labdák kilövődnek ha be van pipálva a **[shoot]**, és leesnek a gravitációnak megfelelően ha nincsen;



(a) Nagyon sok labda



(b) Extrém fraktál

3.14. ábra. A labdák ütközése különböző körülmények között

12. A **labdák** ütközése tárgyakkal valósághűnek tűnik (3.14. ábra);
13. A **labdák** ütközése egymással közepesen valóságosnak tűnik, a kezdőpozíció alatti félgömbhéjban 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("Avgage 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 }
```

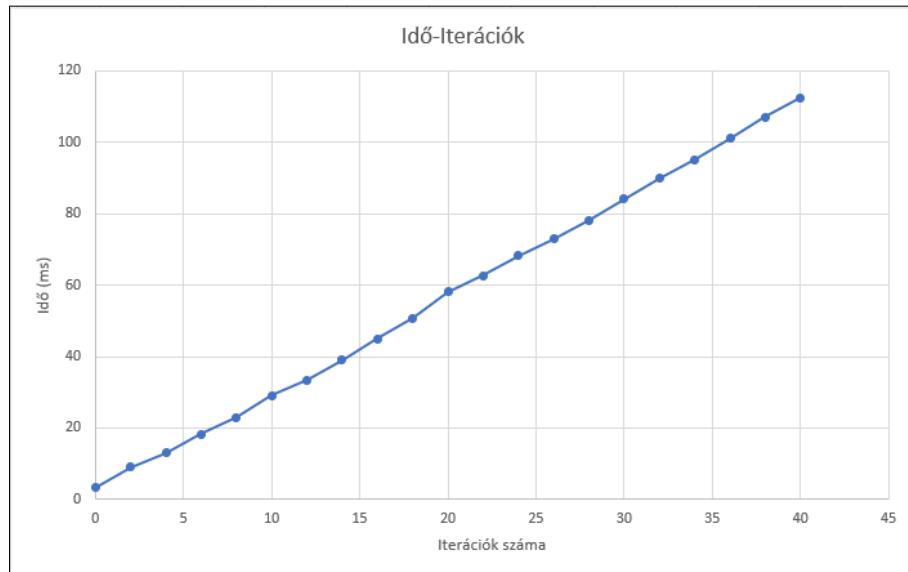
3.4. forráskód. A tesztelést végző kód

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 $1/60s = 16.66ms$ -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

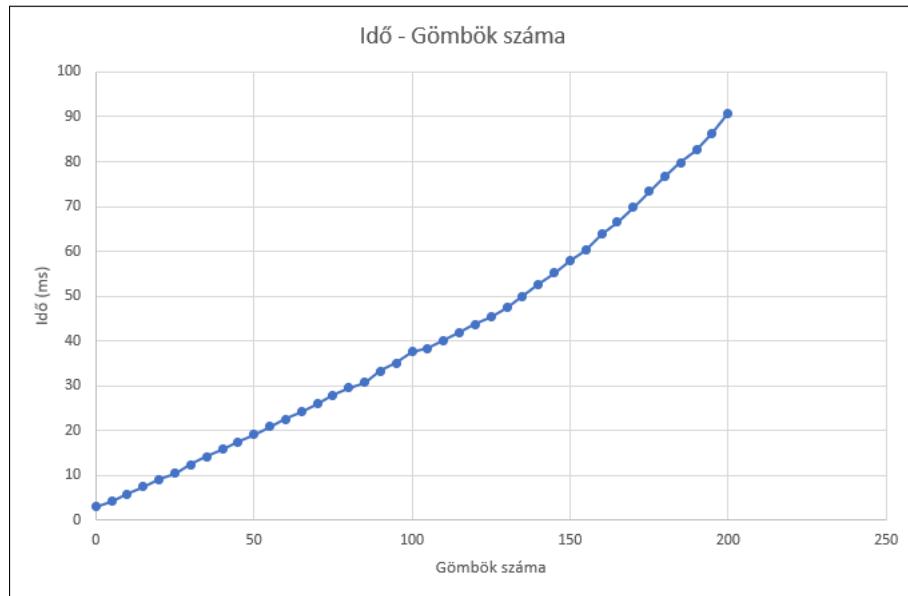
képest. A futás eredményéből készült grafikon a 3.15. ábrán látható.



3.15. ábra. Teljesítményteszt az iterációk számának függvényében

Megfigyelhető a 3.15. ábrán hogy az iterációk számától lineárisan függ a képfritsítési idő. Az adatok alapján az iterációk számának eggyel való növelése átlagosan nagyjából 2.8 ms-os növekedést okoz a renderelési időben a teszteléshez használt konfiguráción. Ebből adódóan nem érdemes nagyon magas iterációs számmal dolgozni, mert hamar szaggatott lehet a kirajzolás.

Ha a 3.4. kód 15. sorát átírjuk **ballCount += 5** -re akkor azt is meg tudjuk vizsgálni hogy az mozgatható gömbök (labdák) száma hogyan hat a képfritsítési időre. Ezen futás eredményéből készült grafikont a 3.16. ábra mutatja.

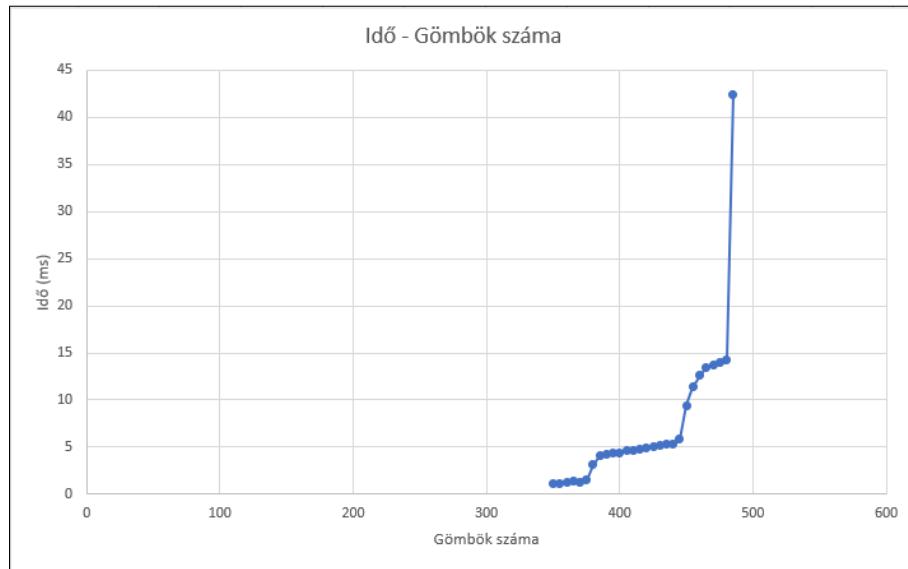


3.16. ábra. Teljesítményteszt a gömbök számának függvényében

Itt is nagyjából lineáris összefüggést tapasztalunk, a labdák számának eggyel való növeléséhez átlagosan 0.4 ms-os növekedés tartozik a kirajzolási időben. Azt tapasztaljuk hogy jóval kevesebb hatása van a gömbök számának növelése a teljesítményre – kb. 1/7-e az iterációhoz képest. A tesztkonfigurációnak 100 labda megjelenítse sem okoz problémát, amíg az iterációk száma kelleően alacsonra van állítva.

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 az előbbi tesztet, akkor megtudhatjuk hogy a fizika kiszámolása mennyire lassította a kirajzolást. A futás eredményének egy részéből készült grafikon a 3.17. 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.



3.17. 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 mind 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 úgy, hogy 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. 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 tételere 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] Bálint Csaba. *Távolságfüggvényekkel definiált felületek interaktív megjelenítése.* https://people.inf.elte.hu/csabix/publications/conference/OTDK_prezi.pdf. (Accessed on 05/30/2020). 2017.

- [10] *Grafika BSc Gyakorlat anyagok – ELTE Számítógépes Grafika.* <http://cg.elte.hu/index.php/grafika-bsc-gyakorlat-anyagok/>. (Accessed on 05/13/2020).
- [11] *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).
- [12] *Python & OpenGL for Scientific Visualization.* <https://www.labri.fr/perso/nrougier/python-opengl/>. (Accessed on 05/24/2020).
- [13] *Spherical coordinate system - Wikipedia.* https://en.wikipedia.org/wiki/Spherical_coordinate_system#Coordinate_system_conversions. (Accessed on 05/26/2020).
- [14] *Inigo Quilez :: fractals, computer graphics, mathematics, shaders, demoscene and more.* <https://www.iquilezles.org/www/articles/distfunctions/distfunctions.htm>. (Accessed on 05/27/2020).

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 megfelelő paraméterekkel látványos fraktálokat kaphatunk	7
2.6.	Bármilyen paraméter esetén képesek a labdák helyes ütközésre	8
2.7.	A „Parameters” feliratú panel alsó harmada	9
2.8.	A [zoom] paraméter hatása a képre (a kamera mozgatása nélkül)	9
2.9.	A labdák helyzete a szóköz lenyomása közben	10
2.10.	A labdák kilövése utáni pillanatkép	10
3.1.	Az SDL logója	13
3.2.	Az OpenGL logója	13
3.3.	A GLM logója	14
3.4.	A raycast technika ábrázolása. [7]	14
3.5.	Sphere tracing: A kamerából kiinduló fénysugár mindenkorának annyit halad előre amekkora a hozzá legközelebb lévő felület távolsága [1]	15
3.6.	A fénymodell különböző komponensei	16
3.7.	A kiindulási pozíció alatt található alakzat néhány labdával	18
3.8.	A CMyApp osztálydiagramja (két részre szedve a hosszúsága miatt)	19
3.9.	A skaláris szorzat előjele és a vektorok közötti szög összefüggése [11]	21
3.10.	NDC (Normalized Device Coordinates) - az ablak koordinátái [12]	23
3.11.	A gCamera osztálydiagramma	24

3.12. Polár koordináták [13]	26
3.13. A paraméterek egy lehetséges kombinációja amikor már éppen csak látható a fraktál, kicsit módosítva a megfelelőkön már nem látható . .	30
3.14. A labdák ütközése különböző körülmények között	31
3.15. Teljesítményteszt az iterációk számának függvényében	33
3.16. Teljesítményteszt a gömbök számának függvényében	34
3.17. Teljesítményteszt a gömbök számának függvényében, kirajzolás nélkül	35

Forráskódjegyzék

3.1.	A sphere tarcing algoritmust megvalósító kód	15
3.2.	A felületi normálist kiszámoló függvény	21
3.3.	Az eye és at vektorok frissítése	24
3.4.	A tesztelést végző kód	32