



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. Tesztelés	10
3.1.1. Működés helyessége	10
3.1.2. Teljesítmény	11
4. Összegzés	15
5. További fejlesztési lehetőségek	16
A. Függelék	18
Irodalomjegyzék	19
Ábrajegyzék	20
Táblázatjegyzék	21
Forráskódjegyzék	22

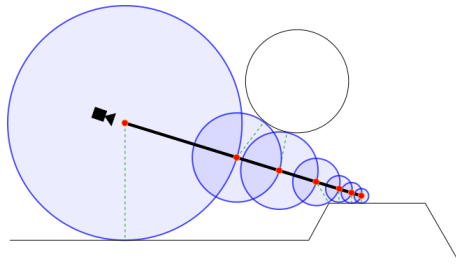
1. fejezet

Bevezetés

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

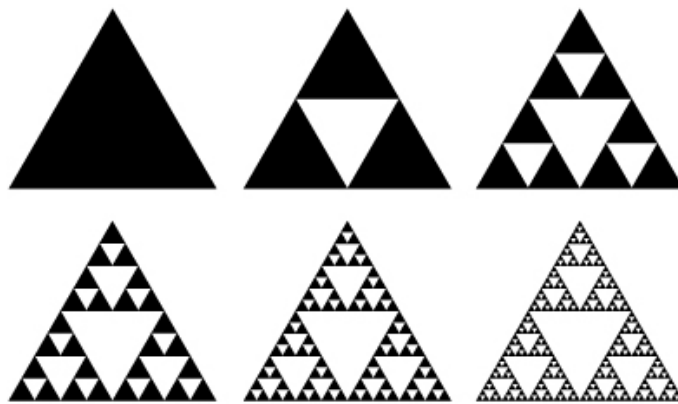
Egy játékmotor feladata hogy leegyszerűsítse a kirajzolást és az objektumok **valóság-hű viselkedését**. Ilyen viselkedés például, ha két szilárd tárgy ütközésekor azt várnánk el hogy azok ne menjenek bele egymásba, hanem inkább ténylegesen ütközzenek és "pattanjanak le" a másikról. Ezen viselkedés kiszámolása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

Szakedolgozatomban azzal foglalkozom hogy hogyan lehet a kirajzolást és az előbb említett valóság-hű 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 mindig csak 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

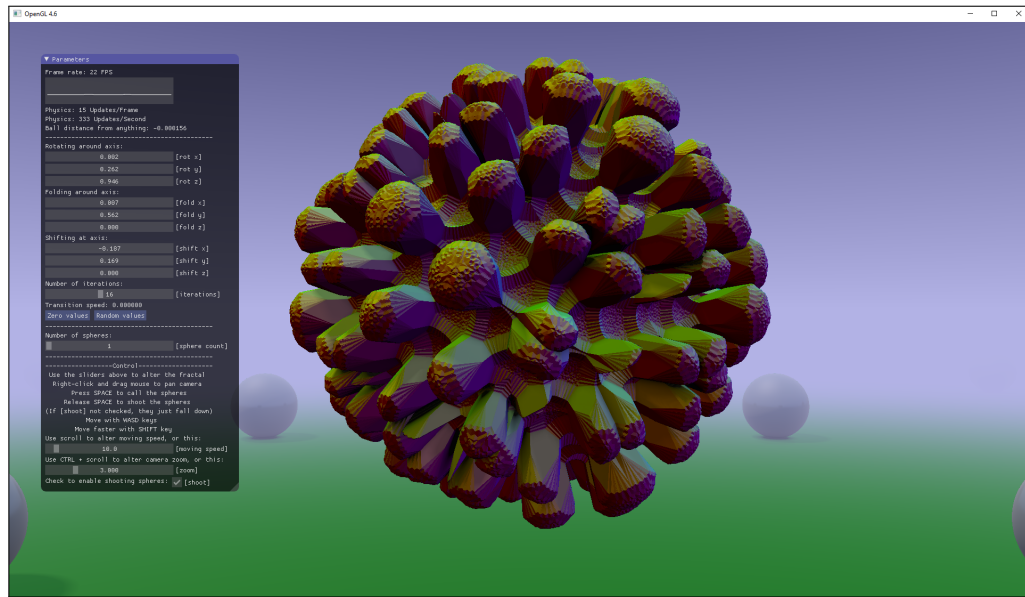
Felhasználói dokumentáció

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 bejárni, 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

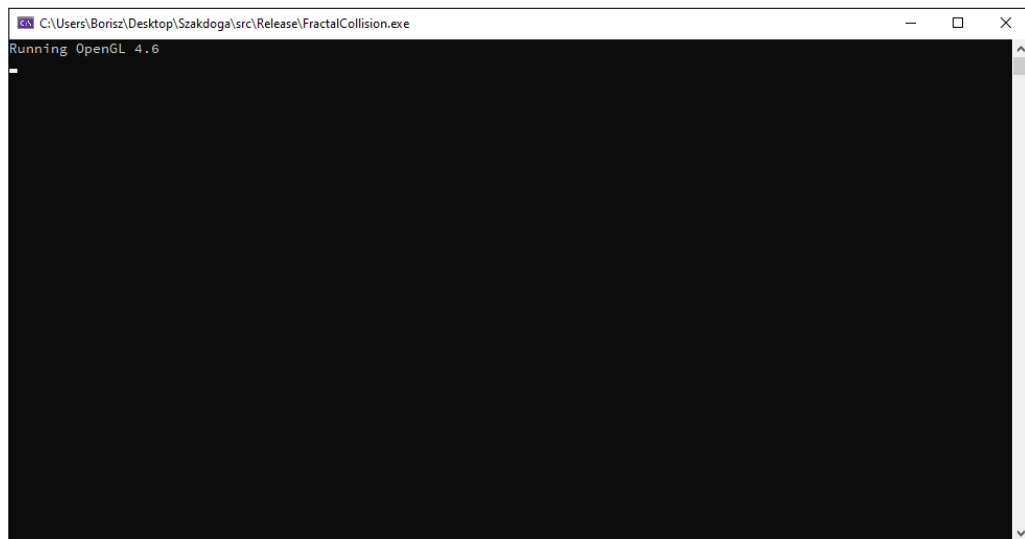
A program sikeres indítása után – melyről a 2.2. fejezetben tudhatunk meg többet – kettő darab ablakkal találjuk szembe magunkat. Ezekről a 2.1. és 2.2. ábrák mutatnak egy-egy képernyőképet.

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 hiba-üzeneteket fogja kiírni. Ezen kívül ez az ablak más funkcióval nem bír.

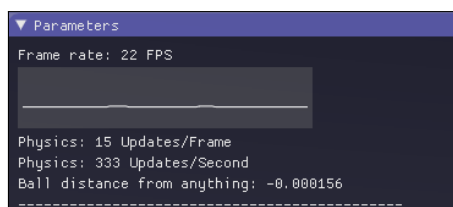


2.2. ábra. Terminálablak

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.

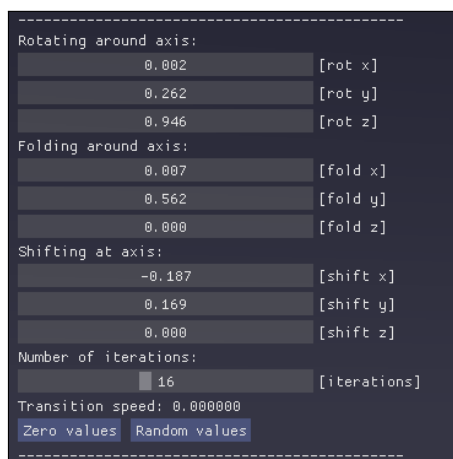


2.3. ábra. A „Parameters” feliratú panel felső harmada

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 létrehozottra 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 $1 \times 1 \times 2$ egység nagyságú téglatesten egymás után többször végrehajtottunk 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 begépett 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).

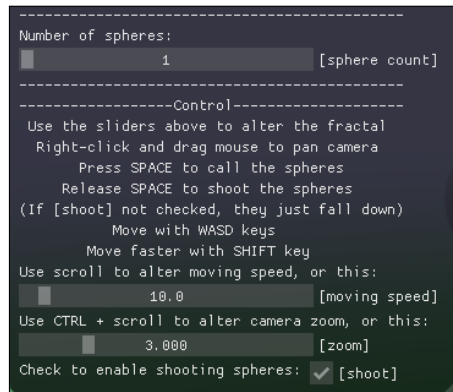
A **[fold x]**, **[fold y]**, **[fold z]** csúszkákkal azt tudjuk szabályozni hogy mennyire legyen elforgatva az adott tengely körül a tükrözősík ami a megadott szöggel (radián) fordul el és tükrözi az alakzatot iterációnként. Itt a három érték 3 különböző tükrözősíkot forgat el a nevükben szereplő tengely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

A **[shift x]**, **[shift y]**, **[shift z]** csúszkák szabályozzák hogy mennyire legyen eltolva az alakzat iterációnként 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 végrehajtván a téglatesten. Ez a paraméter van a legnagyobb hatással a futás sebességére, így gyengébb gépeken nem érdemes nagy értéket beállítani. Ha a képfriességi 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 kapcsolódó érték ami a **“Transition speed:”** felirat után olvasható. A gombok által generált új paramétereit 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 bejárásához szükséges irányítások is. A mozgáshoz 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 + görgő** 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 körvonal mentén egyenletesen elhelyezkedve. A szóköz billentyűt felengedve egyszerre kilövednek a labdák, ha be van pipálva a panel alján a **[shoot]** jelölőnégyzet, ha nincsen bepipálva csak leesnek a gravitációnak 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 egyetlen követelménye a Windows 7 vagy afeletti operációs rendszer. 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 sarokban az ablak bezárás gombját használni. Ha bezárjuk a terminálablakot akkor mindkét 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, így ha újra akar-nánk fordítani akkor a **C:/** helyre csomagoljuk ki a mellékelt OGLPack.zip állományt (ez az OpenGL-hez szükséges fájlokat tartalmazza), majd futtassuk a **subst T: C:/** parancsot. Ezután megnyithatjuk a **.vcxproj** projektfájlt.

3.1. 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

3.1.1. 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 görgővel vagy a csúszkával;

5. Ha egyszerre gyorsítunk **SHIFT**-tel és görgővel, a görgő sebessége felülírja a shift gyorsítását;
6. A fraktál paramétereit át lehet állítani a **csúszkával** és a fraktál ezen értékeknek megfelelően változik;
7. A „Zero values” gomb hatására megközelíti minden érték a nullát;
8. A „Zero values” gomb extrém érték esetén: 1000-re állítva egy csúszkát "nullázás" után az értéke 0.423 lett;
9. A „Random values” gomb valóban véletlenszerű értékeket állít be. Időnként nem látható fraktálokat eredményez;

3.1.2. Teljesítmény

Az alkalmazás erősen GPU igényes. A futás teljesítményét az alábbi kód segítségével teszteltem:

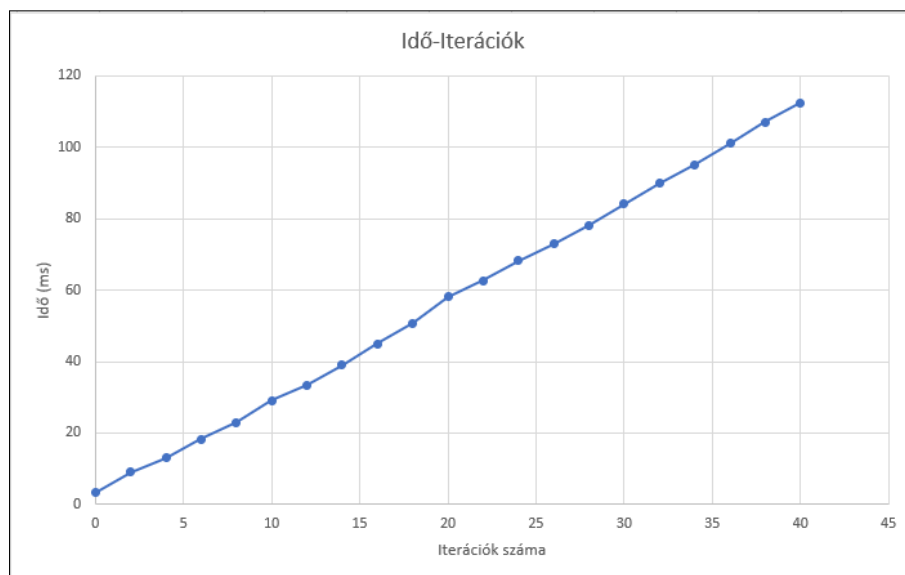
```
1 if (TESTING)
2 {
3     if (delta_time_counter < avg)
4     {
5         delta_time_arr[delta_time_counter] = delta_time;
6         ++delta_time_counter;
7     }
8     else
9     {
10        delta_time_counter = 0;
11        double avg_delta_time = 0.0;
12        for (int i = 0; i < avg; ++i) { avg_delta_time +=
13            delta_time_arr[i]; }
14        avg_delta_time /= avg;
15        printf("Avrage 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 }
```

3.1. 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 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 3.1. ábrán látható.

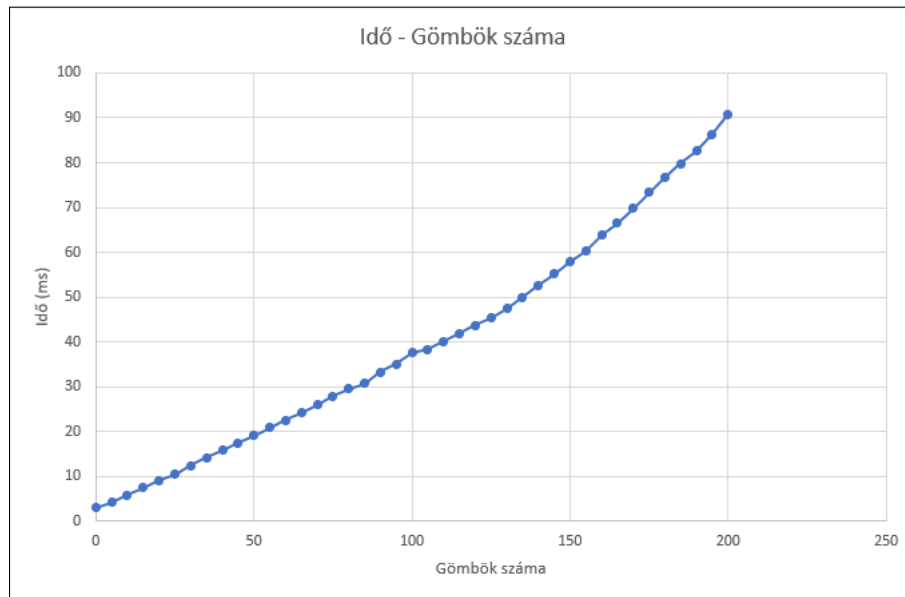


3.1. ábra. Teljesítményteszt az iterációk számának függvényében

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 környezetben 10 FPS alá essen a képfrissítési ráta (vagyis 100 ms fölé megy a frissítési idő), ami már határozottan nem folyamatos megjelenítést jelent.

Ha a 3.1. 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 ?? ábra mutatja.

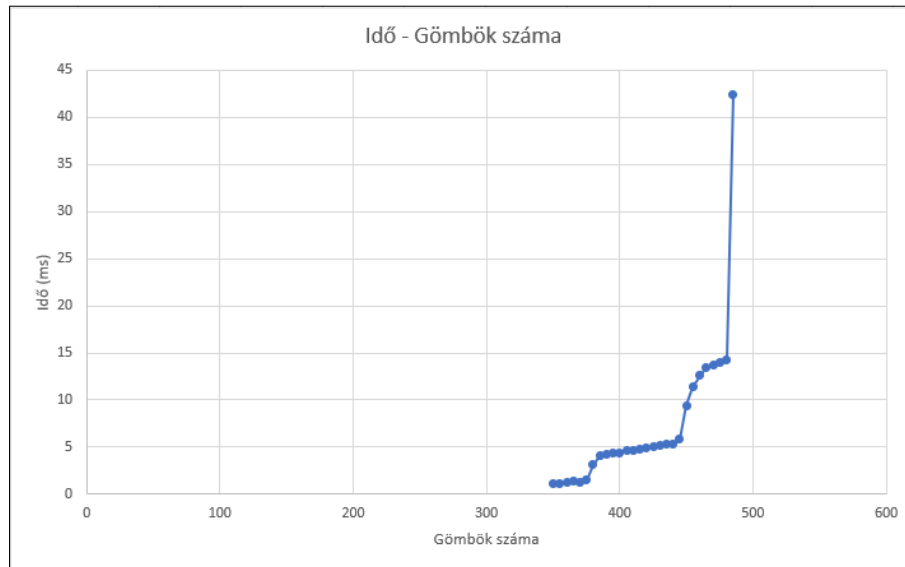


3.2. ábra. Teljesítményteszt a gömbök számának függvényében

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 3.3. 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.



3.3. 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 jelenség pontos forrása ismeretlen, de mivel csak extrém körülmények között jelentkezik, így nem lett sok idő fordítva az ok felkutatására.

4. fejezet

Összegzés

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

A használt fraktálgenerálási módszer az iterációk szabad szabályozásával szemléletesen megmutatja hogy 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.

5. fejezet

További fejlesztési lehetőségek

A legnagyobb és legegységertelműbb 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, míg 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 **valószínűbb 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 vektora megfelelően

súlyozott átlagát és fordítva.

A gömbök **kilövése** lehetne úgyhogy egy (esetleg néhány) labda van közepen amit mindig 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 számnál még jól követhető hogy hogyan hatnak az alakzatra.

Ha a teljesítmény javult, a **fényszámításon** is sokat lehet javítani. Sphere Tracing algoritmussal akár közel fotorealistikus fényeink is lehetnek. Persze találni kell egy kompromisszumot kinézet és gyorsaság között, de vannak jól ismert trükkök a fények szebbé tételére 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 kinézőek és vannak unalmasabbak. A puszta 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.

A. függelék

Függelék

folyamtban...

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 easies - C++ Articles*. <https://jutge.org/doc/cplusplus.com/articles/LyTbqMoL/index.html>. (Accessed on 05/10/2020).

Ábrák jegyzéke

1.1. Sphere tracing: A kamerából kiinduló fénysugár mindig csak 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. Teljesítményteszt az iterációk számának függvényében	12
3.2. Teljesítményteszt a gömbök számának függvényében	13
3.3. Teljesítményteszt a gömbök számának függvényében, kirajzolás nélkül	14

Táblázatok jegyzéke

Forráskódjegyzék

3.1. A tesztelést végző kód	11
---------------------------------------	----