



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

<b>1. Bevezetés</b>	<b>3</b>
<b>2. Felhasználói dokumentáció</b>	<b>5</b>
2.1. Felhasználói felület és funkciók . . . . .	5
2.1.1. A „Parameters” feliratú panel . . . . .	6
2.2. Rendszerkövetelmények és futtatás . . . . .	10
<b>3. Fejlesztői dokumentáció</b>	<b>11</b>
3.1. Fejlesztői eszközök . . . . .	11
3.1.1. Simple DirectMedia Layer (SDL) . . . . .	11
3.1.2. OpenGL API . . . . .	12
3.1.3. Dear ImGui . . . . .	12
3.1.4. GLM . . . . .	13
3.2. Képalkotási módszer . . . . .	13
3.2.1. Fénymodell . . . . .	15
3.3. Megvalósítás . . . . .	16
3.3.1. CMyApp osztály . . . . .	16
3.4. Tesztelés . . . . .	18
3.4.1. Működés helyessége . . . . .	18
3.4.2. Teljesítmény . . . . .	19
<b>4. Összegzés</b>	<b>23</b>
<b>5. További fejlesztési lehetőségek</b>	<b>24</b>
<b>A. Függelék</b>	<b>26</b>
<b>Irodalomjegyzék</b>	<b>27</b>

<b>Ábrajegyzék</b>	<b>28</b>
<b>Táblázatjegyzék</b>	<b>29</b>
<b>Forráskódjegyzék</b>	<b>30</b>

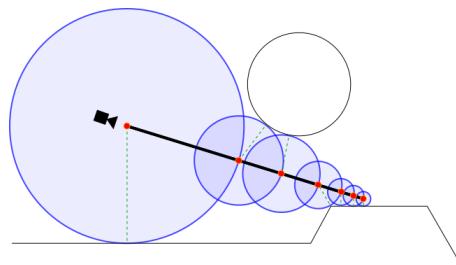
# 1. fejezet

## 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.

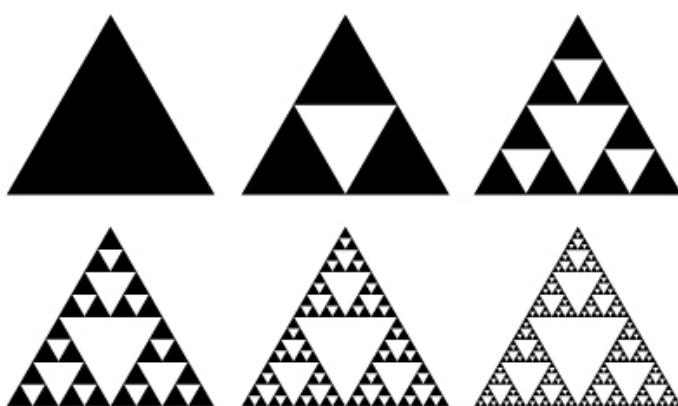
Egy játékmotor feladata hogy leegyszerűsítse a kirajzolást és az objektumok **valósághű viselkedését**. Ilyen viselkedés például, ha két szilárd tárgy ütközésekor azt várunk el hogy azok ne menjenek bele egymásba, hanem inkább ténylegesen ütközzenek és "pattanjanak le" a másikról. Ezen viselkedés kiszámolása 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 annyi halad előre amekkora a hozzá legközelebb lévő felület távolsága [1]

Hogyan valósítom ezt meg? A kirajzoláshoz **Sphere tracing** módszert (3.5. 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 valamelyen 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 csoporthzával 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

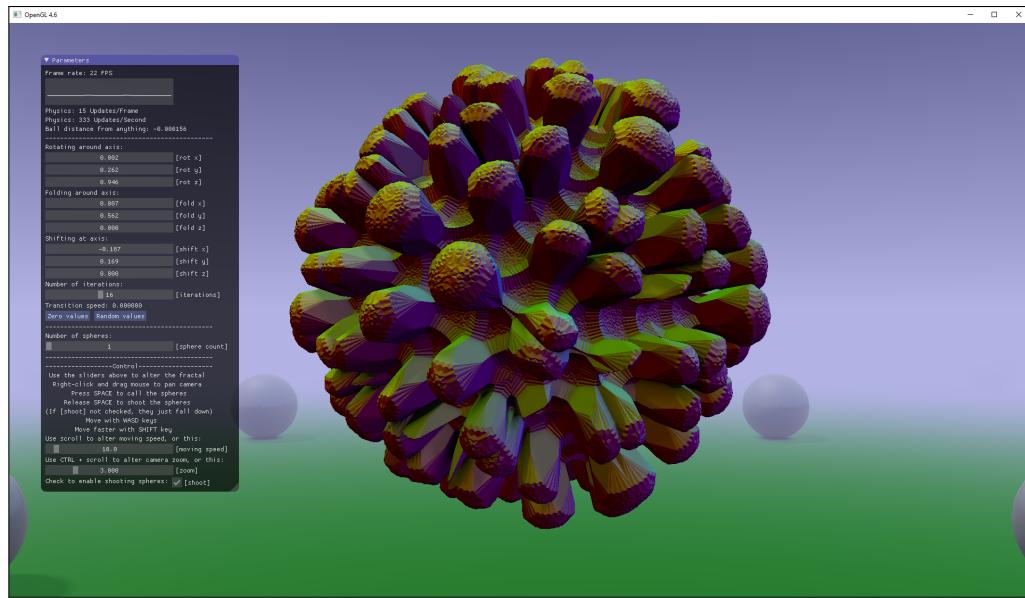
# Felhasználói dokumentáció

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

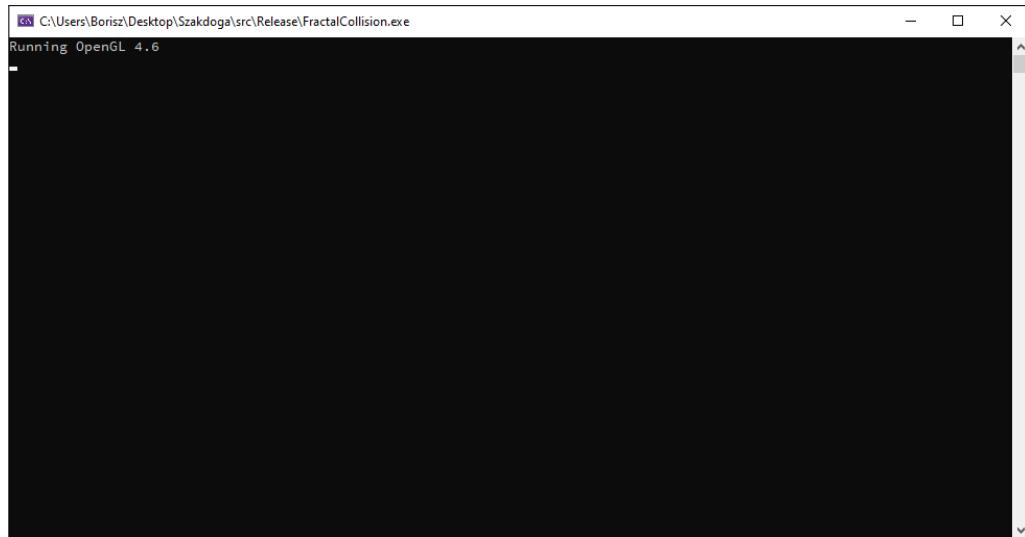
A program sikeres indítása után – melyről a 2.2. fejezetben tudhatunk meg többet – kettő darab ablakkal találjuk szembe magunkat. Ezekről a 2.1. és 2.2. ábrák mutatnak egy-egy képernyőképet.

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.

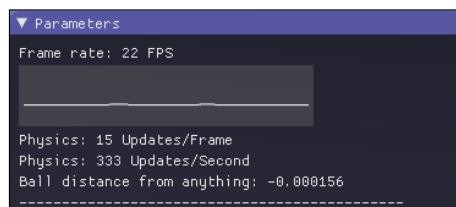


2.2. ábra. Terminálablak

### 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.

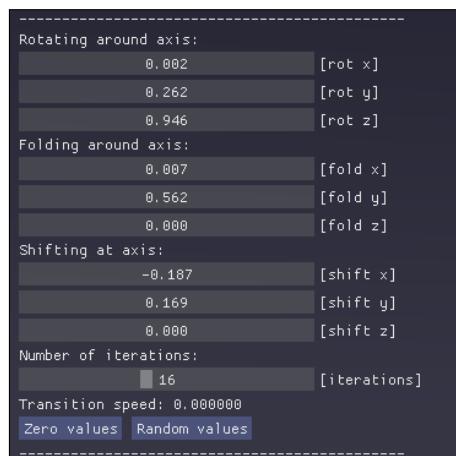


2.3. ábra. A „Parameters” feliratú panel felső harmada

A panel legtetején (2.3. ábra) találhatjuk a „**Frame rate:**” felirat után az aktuális képfrissítési rátát képkocka/másodperc mértékegységen, 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 létrehozott 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.

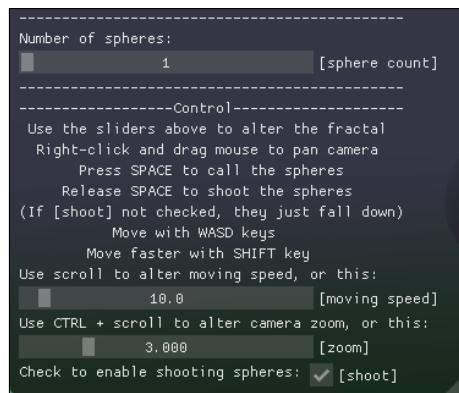
A **[shift x]**, **[shift y]**, **[shift z]** csúszkák szabályozzák hogy mennyire legyen eltolva az alakzat iterációinként az X, Y és Z tengely mentén.

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 közelíti<sup>1</sup>, 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.

---

<sup>1</sup>Az 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 mozgásra a **WASD** billentyűket és az egeret lenyomott jobb egérgombbal<sup>2</sup> 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 + görög** 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.

---

<sup>2</sup>Erre azért volt szükség mert a paraméterek beállításához is az egeret kell használni, muszáj volt elkülöníteni valahogyan 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 bezártuk 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.

#### 3.1.1. Simple DirectMedia Layer (SDL)

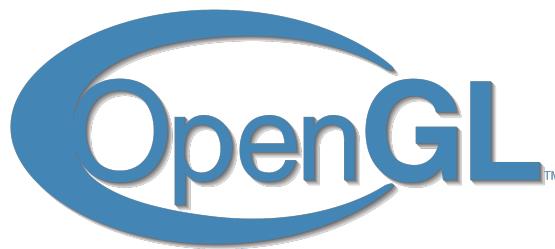
A Simple DirectMedia Layer (SDL) könyvtár egy crossplatform multimédiás könyvtár, ami alacsony szintű, hatékony hozzáférést ad audio, bemeneti (egér, billentyűzet, joystick), valamint grafikus (OpenGL-en keresztül GPU-hoz) eszközökhöz. Az alkalmazásban az SDL 2.0 van használva. [3]



3.1. ábra. Az SDL logója

### 3.1.2. 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

### 3.1.3. 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]

### 3.1.4. GLM

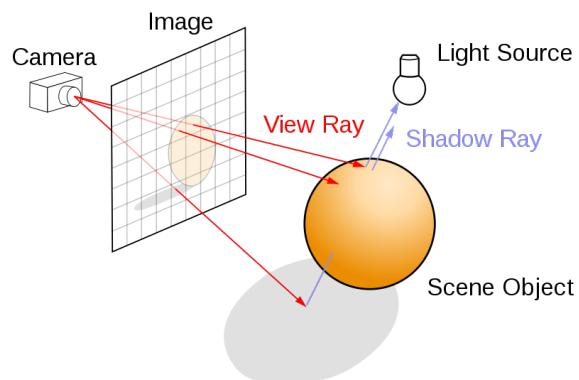
Az OpenGL Mathematics egy headerben importálható C++ könyvtár mely a GLSL specifikációira szabva (elnevezések, funkciók) tartalmaz számtalan matematikai funkciót. Segítségével a GLSL-ben használatos funkciók és típusok C++-ban is használhatóak. [6]



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. [FileRayt97:online]

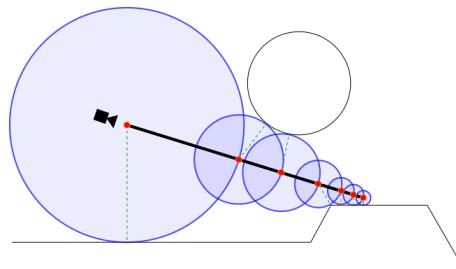
A **sphere tracing** algoritmus egy speciális esete az úgynevezett **raymarching** algoritmusnak. Működéséhez a kirajzolni kívánt objektumokat távolságfüggvényekkel kell reprezentálni, mely a virtuális tér bármely pontjáról képes meghatározni hogy milyen messze van az objektum felületétől. Az algoritmus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

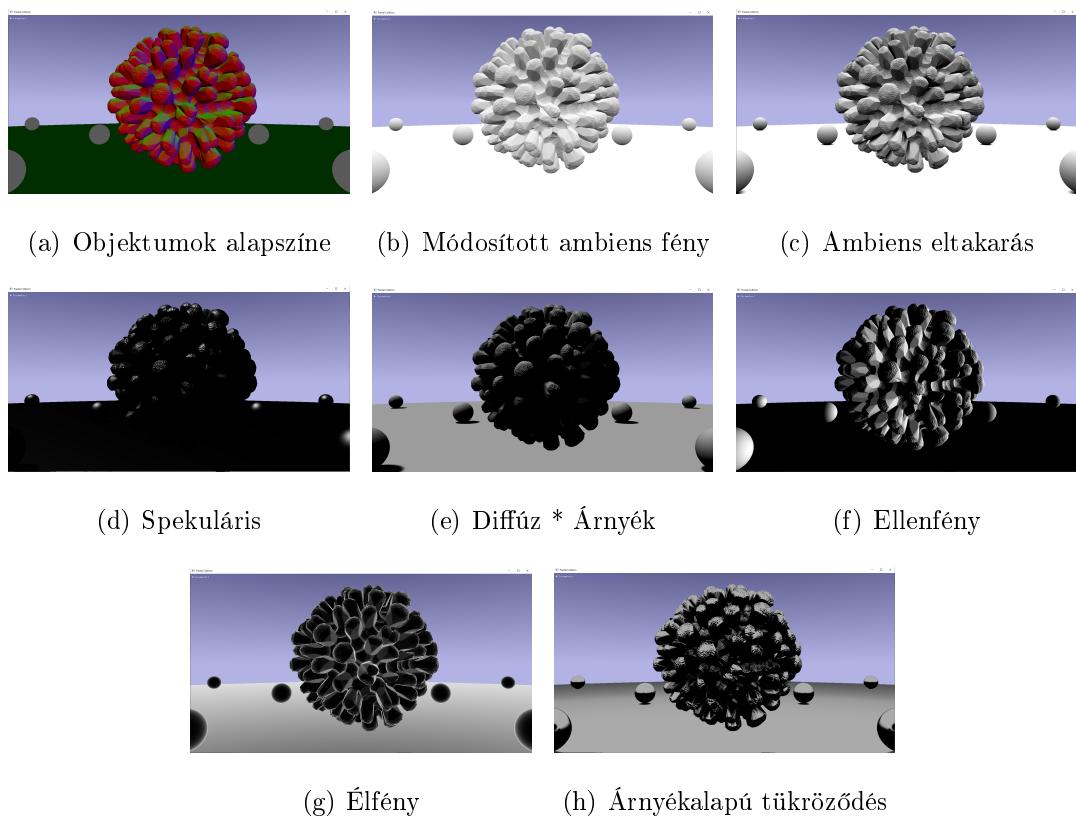
Az alkalmazásban teljesítményjavítás érdekében másik feltétel ( $dS < 0.01 * dist / MAX\_DIST$ ) lett használva a felületi távolsághoz, mely számításba veszi hogy mennyire messzire vagyunk a kamerától. Ha messzebb vagyunk

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. Ezen fénymodell komponensei a 3.6. ábrán láthatóak. Ezek különböző színenkénti súlyozással vett összege alkotja végső színáranyalatot, melyre még egy gamma korrekció és egy ködszerű effektus került – ez utóbbi arra szolgál hogy a viszonylag kicsi maximális távolság leplezzé.



3.6. ábra. A fénymodell különböző komponensei

### 3.3. Megvalósítás

A program írása során a Számítógépes Grafika BSc gyakorlat tárgy honlapjának [7] projektjei nyújtottak kiindulási alapot. 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 bemene- teit, szimulálja a fizikát és továbbítja a shaderbe a paramétereket hogy megtörténjen a kirajzolás. A fontosabb függvények külön részletezésre kerülnek.

##### **bool Init()**

Ebben inicializálódik minden aminek kell. Definiálunk két darab háromszöget melyek egy téglalapot alkotva lefedik majd a teljes ablakot. Ezt a téglalapot 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 darabszámától függő forgatási mátrix felhasználásával tudjuk a pozíójukat 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 sebességül. Ezáltal minél messzebb van a pozíótól

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 ütközünk.

A helyes **viselkedés megállapításához** ismerünk kell az ütközés pontjában a felületi normálist. A normális 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.

```

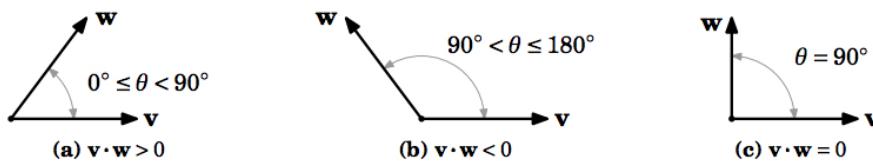
1 float RayMarch(vec3 ro, vec3 rd) {
2     float dist=0.0;
3     for(int i=0; i<MAX_STEPS; i++) {
4         vec3 p = ro + rd*dist;
5         float dS = GetDist(p);
6         dist += dS;
7         if(dist>MAX_DIST || dS<SURF_DIST ) break;
8     }
9     return dist;
10 }
```

3.2. forráskód. A felületi normálist kiszámoló függvény

Ezután már csak a kapott normálisra kell tükrözniünk a vizsgált labdánk sebességvektorát és a megfelelő komponenseit némileg 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 belevísszük 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.



3.7. ábra. A skaláris szorzat előjele és a vektorok közötti szög összefüggése

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 sebességevektor, hiszen akkor állandóan oda-vissza tükröződne és sosem jutna ki a labda. Erre megoldás hogy csak akkor tükrözzük a sebességevektort 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)
```

Megtörténik a kirajzolás és a shader meghívása. Paraméterként megkapja az ablak méreteit.

## 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 viselkedésekkel 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 értéknek 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 "nulázás" után az értéke 0.423 lett;
9. A „**Random values**” gomb valóban véletlenszerű értékekkel állít be. Időnként nem látható fraktálokat eredményez;

### 3.4.2. 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;

```

```

14     printf("Averge of delta time: %f ms    Iterations: %d    Number
15         of spheres: %d \n", avg_delta_time*1000, iterations,
16         ballCount);
17     iterations += 2;
}
}

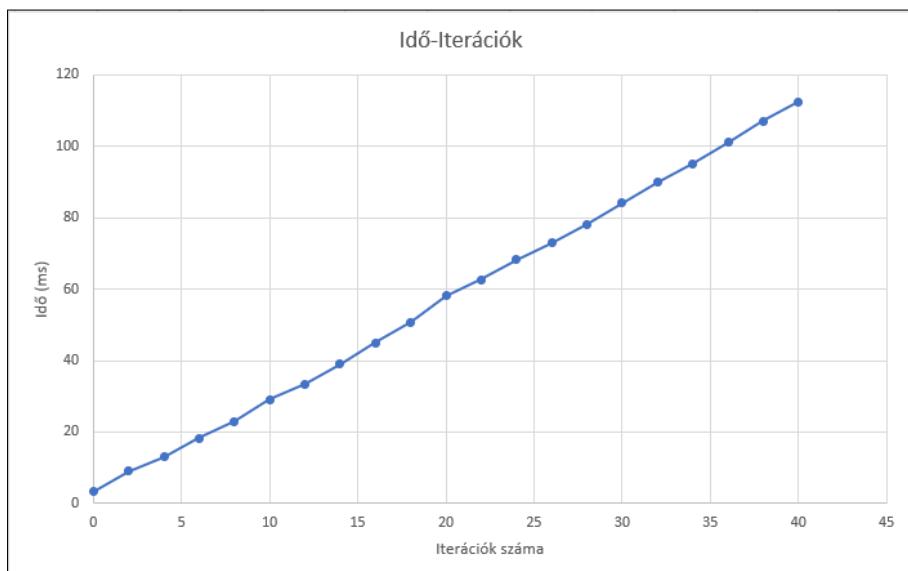
```

### 3.3. 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  $1/60$  s = 16.66 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 3.8. ábrán látható.

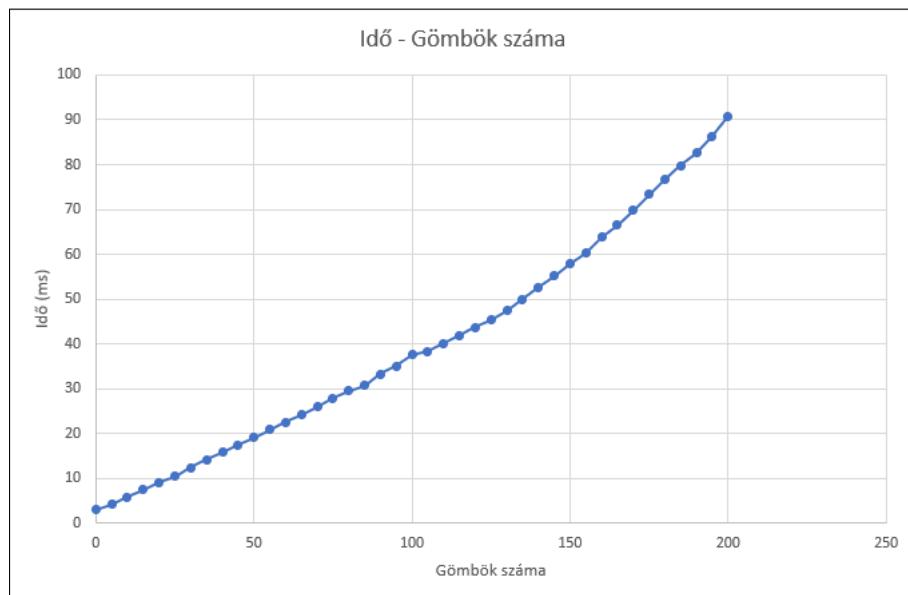


3.8. ábra. Teljesítményteszt az iterációk számának függvényében

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 3.3. 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.

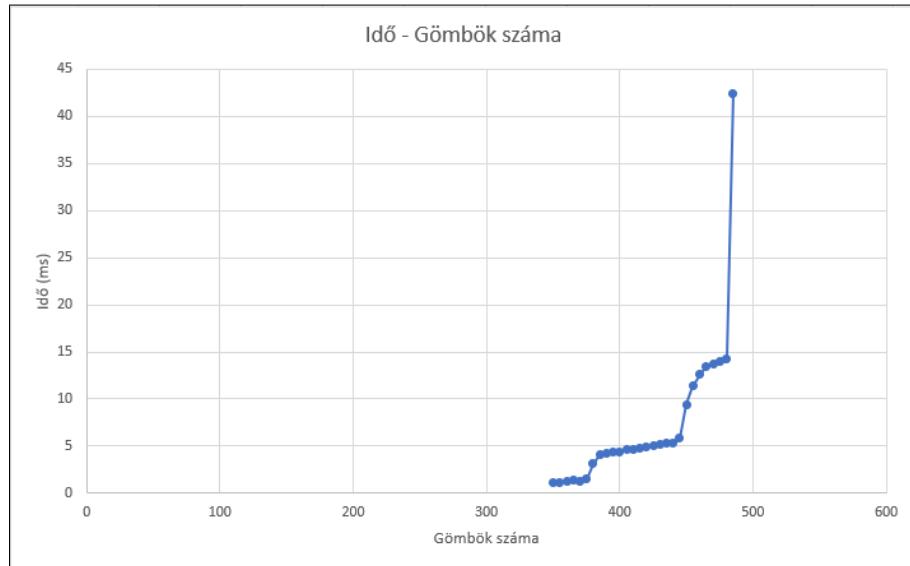


3.9. ábra. Teljesítményteszt a gömbök számának függvényében

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 3.10. 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.



3.10. 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 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.

## 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.

## 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 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](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] *Grafika BSc Gyakorlat anyagok – ELTE Számítógépes Grafika.* <http://cg.elte.hu/index.php/grafika-bsc-gyakorlat-anyagok/>. (Accessed on 05/13/2020).

# Ábrák jegyzéke

1.1.	Sphere tracing: A kamerából kiinduló fénysugár minden csak annyit halad előre amekkora a hozzá legközelebb lévő felület távolsága [1] . . . . .	3
1.2.	Példa egy IFS-re: a Sierpiński háromszög néhány iterációja [2] . . . . .	4
2.1.	Fő programablak . . . . .	6
2.2.	Terminálablak . . . . .	6
2.3.	A „Parameters” feliratú panel felső harmada . . . . .	7
2.4.	A "Parameters" feliratú panel középső harmada . . . . .	7
2.5.	A "Parameters" feliratú panel alsó harmada . . . . .	9
3.1.	Az SDL logója . . . . .	12
3.2.	Az OpenGL logója . . . . .	12
3.3.	A GLM logója . . . . .	13
3.4.	A raycast technika ábrázolása. [ <a href="#">FileRayt97:online</a> ] . . . . .	13
3.5.	Sphere tracing: A kamerából kiinduló fénysugár minden csak annyit halad előre amekkora a hozzá legközelebb lévő felület távolsága [1] . . . . .	14
3.6.	A fénymodell különböző komponensei . . . . .	15
3.7.	A skaláris szorzat előjele és a vektorok közötti szög összefüggése . . . . .	18
3.8.	Teljesítményteszt az iterációk számának függvényében . . . . .	20
3.9.	Teljesítményteszt a gömbök számának függvényében . . . . .	21
3.10.	Teljesítményteszt a gömbök számának függvényében, kirajzolás nélkül . . . . .	22

# Táblázatok jegyzéke

# Forráskódjegyzék

3.1.	A sphere tarcing algoritmust megvalósító kód	14
3.2.	A felületi normálist kiszámoló függvény	17
3.3.	A tesztelést végző kód	19