Szakdolgozat

Miskolci Egyetem

Ütközésvizsgálat automatikus optimalizálása térbeli modellekhez

Készítette:

Szöllősi János

Programtervező informatikus

Témavezető:

Dr. Nagy Noémi

Miskolc, 2023

Miskolci Egyetem
Gépészmérnöki és Informatikai Kar
Alkalmazott Matematikai Intézeti Tanszék **Szám:**

Szakdolgozat Feladat

Szöllősi János (BC6X4X) Programtervező informatikus jelölt részére.

A szakdolgozat tárgyköre: Geometria, Optimalizálás

A szakdolgozat címe: Ütközésvizsgálat automatikus optimalizálása térbeli modellekhez

A feladat részletezése:

A háromszögek metszésének számítása a számítógépi grafikában, térbeli modellezésben, szimulációkban egy gyakori probléma. A szakdolgozat az ehhez kapcsolódó számításokat, térpartícionálási és egyéb optimalizálási módszereket vizsgálja.

A cél egy olyan függvénykönyvtár elkészítése, amely egy adott modell alapján létrehoz egy olyan struktúrát/objektumot, mely segítségével az ütközésdetektálás hatékonyan megoldható. A dolgozat bemutatja az elkészített algoritmusok működését, adatszerkezeteit, becslést ad a számítások idő- és tárigény komplexitására. A függvénykönyvtár C programozási nyelven készül, melyhez szemléltetés céljából OpenGL megjelenítés is tartozik.

Témavezető: Dr. Nagy Noémi (egyetemi adjunktus)

A feladat kiadásának ideje: 2023. március 13.

. . . . . .
szakfelelős

Eredetiségi Nyilatkozat

Alulírott **Szöllősi János**; Neptun-kód: `BC6X4X` a Miskolci Egyetem Gépészmérnöki és Informatikai Karának végzős Programtervező informatikus szakos hallgatója ezennel büntetőjogi és fegyelmi felelősségem tudatában nyilatkozom és aláírásommal igazolom, hogy *Ütközésvizsgálat automatikus optimalizálása térbeli modellekhez* című szakdolgozatom saját, önálló munkám; az abban hivatkozott szakirodalom felhasználása a forráskezelés szabályai szerint történt.

Tudomásul veszem, hogy szakdolgozat esetén plágiumnak számít:

- szószerinti idézet közlése idézőjel és hivatkozás megjelölése nélkül;
- tartalmi idézet hivatkozás megjelölése nélkül;
- más publikált gondolatainak saját gondolatként való feltüntetése.

Alulírott kijelentem, hogy a plágium fogalmát megismertem, és tudomásul veszem, hogy plágium esetén szakdolgozatom visszautasításra kerül.

Miskolc, ...év ..... .hó ..... .nap

.....

Hallgató

1.

szükséges (módosítás külön lapon)

A szakdolgozat feladat módosítása

nem szükséges

....

dátum témavezető(k)

2. A feladat kidolgozását ellenőriztem:

témavezető (dátum, aláírás): konzulens (dátum, aláírás):

....

....

....

3. A szakdolgozat beadható:

....

dátum témavezető(k)

4. A szakdolgozat ..... szövegoldalt

..... program protokollt (listát, felhasználói leírást)

..... elektronikus adathordozót (részletezve)

.....

..... egyéb mellékletet (részletezve)

.....

tartalmaz.

....

dátum témavezető(k)

5.

bocsátható

A szakdolgozat bírálatra

nem bocsátható

A bíráló neve: .....

....

dátum szakfelelős

6. A szakdolgozat osztályzata

a témavezető javaslata: .....

a bíráló javaslata: .....

a szakdolgozat végleges eredménye: .....

Miskolc, ...

a Záróvizsga Bizottság Elnöke

Tartalomjegyzék

1. fejezet

Bevezetés

A szakdolgozat célja egy új függvénykönyvtár létrehozása, amely létrehoz automatikusan egy térbeli "hitboxot"[1.1] a beimportált modellekhez, ezzel lehetővé téve a modellek közötti ütközések vizsgálatát. A szakdolgozathoz készített program C [5] nyelven íródott OpenGL [3] és SDL2 [4] függvénykönyvtárak segítségével. A program célja, hogy a modelleket felbontsa atomi szintre (háromszögekre), illetve a háromszögek metszéspontjainak kiszámításával pontos ütközésvizsgálat valósuljon meg térpartícionálás [2] segítségével.

Az ütközésvizsgálat, illetve "hitboxok"[1.1] használata rendkívül fontos a számítógépes grafikában és a játékfejlesztésben. Játékok és fizikai szimulációk fejlesztése során rendkívül fontos, hogy a modellek ütközésvizsgálata megbízható és pontos legyen. A szakdolgozatban szereplő algoritmus a pontosságot célozza meg.

Az atomi szintű ütközésvizsgálat során nagy pontosságot érhetünk el, ezzel valósághű és precíz számításokat, szimulációkat végezhetünk el. A szakdolgozat során részletesen bemutatásra kerül a függvénykönyvtár tervezése, illetve implementációja.

A fejezetek végigvezetnek a függvénykönyvtár alapvető működésétől, a részletes implementáción át az optimalizációs lehetőségekig, valamint a függvénykönyvtár használatára is kitérnek. A szakdolgozat eredménye egy olyan függvénykönyvtár, amely hozzájárul a játékfejlesztés, illetve számítógépes grafika lehetőségeinek kibővítéséhez, ezzel segítve a fejlesztőket a kidolgozott hatékony ütközésvizsgálati megoldással.

1.1. definíció (Hitbox). A "hitbox" a számítógépes programozásban egy objektum határait jelöli, amely fontos az ütközések és interakciók szempontjából. Általában láthatatlan testként veszi körül az objektumot, és határozza meg, hogy mikor következik be érintkezés más objektumokkal.

2. fejezet

A szakdolgozat koncepciója

A feladat fő problémája a háromszögek metszéspontjának kiszámítása a térben. Ez sajnos nem egy egyszerű feladat, programozás, illetve erőforrásigény terén nem elhanyagolható probléma.

A valóságban az emberi gondolkodásnak egyszerűnek tűnhet eldönteni, hogy két háromszög metszi-e egymást, vagy sem. Programozás, illetve matematika részéről viszont sokkal nehezebb. Rengeteg számítást kell végeznünk ahhoz, hogy megbizonyosodjunk a háromszögek metszéséről.

2.1. Irodalomkutatás

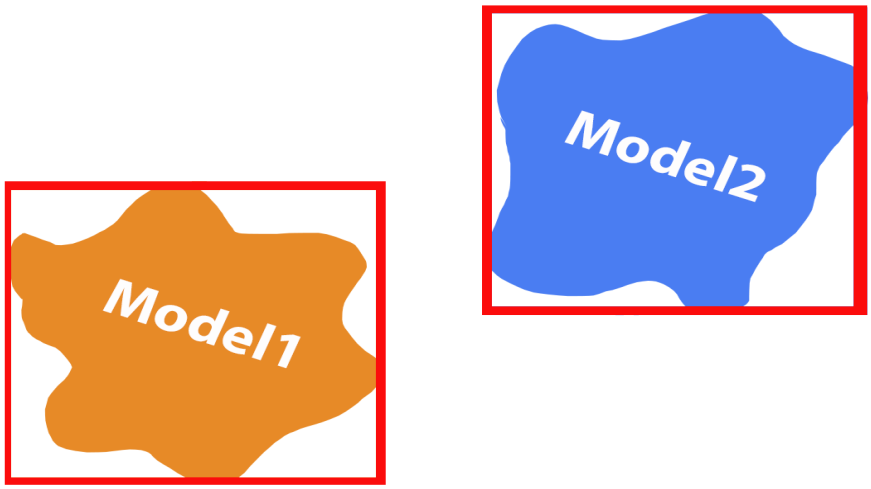
A metszéspontok számításhoz legmegfelelőbbnek David Eberly 1999-es kutatását, azon belül a 4.1 Separation of Triangles [1] szakaszt találtam. A dokumentáció tökéletesen elmagyarázza a matematikai képletek elemeit, azok használatát, lépéseit. Ezek mind táblázatba szedve találhatóak. A pontos magyarázat a 3. fejezetben olvasható. Emellett rengeteg különböző módszer található az interneten térbeli modellek ütközésének vizsgálatához.

2.1.1. Négyzetes elválasztás

A négyzetes elválasztás módszere (Quadratic Separation) egy olyan algoritmus, amelyet számítógépes játékok, illetve számítógépi grafikában közkedvelten használnak ütközések vizsgálatára. A módszer segítségével meghatározhatjuk, hogy két vagy több modell metszi-e egymást vagy sem. Az ötlet lényege, hogy minden modellt egy kocka alakú "dobozokkal" vesszük körbe, amelyet a legkisebb négyszögű terület alapján határozzuk meg. Ekkor az ütközést csak akkor kell ellenőrizni, ha a meghatározott "dobozok" metszik egymást.
Lépései:

- Minden modellhez rendelünk egy "dobozt", amely a legkisebb területen határozza meg a modellt. Ez a "doboz" könnyen meghatározható 2 ponttal a térben. A modellekhez társított területet a 2.1 ábrán láthatjuk.
- Ellenőrizzük a "dobozokat", hogy metszi-e egymást vagy sem.
- Ha két vagy több "doboz" metszi egymást, akkor kerül sor a pontos ütközésvizsgálatra (jelen szakdolgozat esetében ez háromszögek metszéspontjának számítása).

Az előnye a módszernek, hogy a bonyolult modelleket leírhatjuk először egyszerű geomatriai alakzattal, így az ütközésvizsgálat ideje nagyban csökken. Hátránya, hogy az algoritmus csak közelíti az eredményt, ezért fontos egyéb algoritmusokat alkalmazni a pontos ütközésvizsgálathoz.



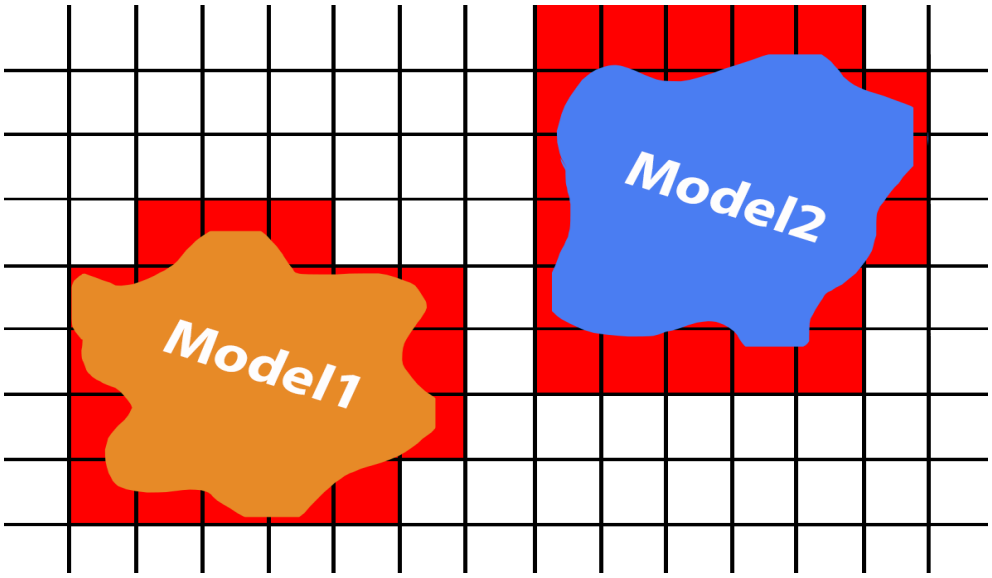
2.1. ábra. Doboz rendelése modellekhez síkban ábrázolva

2.1.2. Voxel-alapú ütközésvizsgálat

A voxel-alapú módszer egy hatékony módszer, amely a (háromdimenziós) teret felosztja kisebb, téglalap alakú részekre. Ezeket a részeket voxelek-nek nevezzük. Ezekkel a voxelekkel közelítjük a modelleket.
Lépései:

- Voxel rácsot hozunk létre. Ezzel a ráccsal osztjuk fel a teret kisebb voxelekre. Ezt a módszert használhatjuk bármilyen dimenzióban.
- A modelleket voxelekkel közelítjük, amelyek a voxel rácsban helyezkednek el. A pontosság módosítható a voxelek mérete alapján. Minden voxelt megjelöljük az alapján, hogy tartalmaz-e modellt vagy sem. A 2.2 ábra jól szemlélteti a modelleket körülvevő aktív voxel cellákat.
- Ütközések vizsgálatához azt ellenőrizzük, hogy a két modellt tartalmazó voxel metszi-e egymást vagy sem. Ha egy aktív voxel metszi a másik aktív voxelt, akkor a modellek ütközhetnek egymással. Ezután szükséges egyéb ütközésvizsgálat a pontosításhoz.

Előnye a módszernek, hogy a használata egyszerű, a pontosság könnyen módosítható. Nagyobb tér esetén hatékony megoldás lehet. Az ütközésvizsgálat csak az aktív voxelek esetén történik meg, így elkerülhetőek a fölösleges számítások. Hátránya, hogy nehéz pontosan meghatározni a voxel rács méretét. Túl nagy méret esetén az ütközésvizsgálat nem lesz elég pontos, kis méret esetén a számítások költségesek lehetnek.



2.2. ábra. A tér felbontása voxelekre a síkban

2.1.3. Egyenletek alkalmazása

Az egyenletek alkalmazásának alapötlete, hogy matematikai egyenletek segítségével határozza meg a modellek mozgását, illetve ütközéseinek érzékelését.
Lépései:

- Felállítunk egy matematikai modellt, amely leírja a modellek mozgását.
- Definiáljuk az ütközések vizsgálatához szükséges szabályokat, feltételeket. Ezek a feltételek lehetnek egyszerűbbek, vagy komplexebbek, például távolságok, sebességek, fizikai tulajdonságok, formák, méretek.
- A meghatározott modell és a feltételek alapján ellenőrizzük az ütközésvizsgálatot.
- Gyakran szükség van az egyenletek alkalmazására numerikus módszerekre, például differenciálegyenletek megoldása esetén.

Előnye a módszernek, hogy pontos fizikai szimulációkat érzékelhetünk. Hátránya, hogy sok esetben magas a számítási igénye, ezért nem alkalmas valós idejű szimulációkra, illetve magas modellszám esetén sem előnyös.

3. fejezet

Ütközések számítása

A háromszöget metszéspontjának számításához elsősorban szükségünk van 2 háromszögre, mint input. Ezek csúcspontjait jelöljük az $\mathbf{A}_0$, $\mathbf{A}_1$, $\mathbf{A}_2$, illetve $\mathbf{B}_0$, $\mathbf{B}_1$, $\mathbf{B}_2$ pontokkal. A csúcspontok segítségével kiszámíthatjuk a háromszögek éleit. Ezek lesznek a $\mathbf{C}_0$, $\mathbf{C}_1$, $\mathbf{C}_2$, illetve $\mathbf{E}_0$, $\mathbf{E}_1$, $\mathbf{E}_2$ élek:

$$\mathbf{C}_0 = \mathbf{A}_1 - \mathbf{A}_0, \mathbf{C}_1 = \mathbf{A}_2 - \mathbf{A}_0, \mathbf{C}_2 = \mathbf{C}_1 - \mathbf{C}_0.$$

Az élek segítségével kiszámíthatjuk a háromszögek normál vektorait. Ezek lesznek a $\mathbf{D}$, illetve $\mathbf{F}$ vektorok:

$$\mathbf{D} = \mathbf{C}_0 \times \mathbf{C}_1, \mathbf{F} = \mathbf{E}_0 \times \mathbf{E}_1.$$

A számítások megkönnyítéséhez kiszámítjuk az eltolásvektort is:

$$\mathbf{G} = \mathbf{B}_0 - \mathbf{A}_0.$$

Ezen adatok szolgálják az alapokat, amelyekből további számításokat végzünk. A metszés eldöntéséhez a következő univerzális képletet használjuk:

$$\begin{aligned}
\mathbf{H}_1 &= \mathbf{L} * \mathbf{C}_0, \\
\mathbf{H}_2 &= \mathbf{L} * \mathbf{C}_1, \\
\mathbf{I}_0 &= \mathbf{L} * \mathbf{G}, \\
\mathbf{I}_1 &= \mathbf{I}_0 + \mathbf{L} * \mathbf{E}_0, \\
\mathbf{I}_2 &= \mathbf{I}_0 + \mathbf{L} * \mathbf{E}_1.
\end{aligned}$$

Minden sor 1-1 számítást jelent. Minden számítás után ellenőriznünk kell, hogy a két adott háromszög metszi-e egymást, vagy sem. Erre a következő képletet használjuk: Ha

$$\mathbf{min(H)} > \mathbf{max(I)} \ \mathbf{vagy} \ \mathbf{max(H)} < \mathbf{min(I)},$$

akkor a két háromszögre biztosan mondható, hogy nem metszik egymást. Ez esetben nem szükséges további számításokat végezni. Amennyiben nemleges választ kapunk, akkor tovább kell számítanunk minden sort. Ha az utolsó sor esetén sem kapunk pozitív választ, akkor kimondhatjuk, hogy a két háromszög metszi egymást. A pontos matematikai számításokat a 3.1 táblázat mutatja.

L	$\mathbf{H}_1$	$\mathbf{H}_2$	$\mathbf{I}_0$	$\mathbf{I}_1$	$\mathbf{I}_2$
D	0	0	D*G	I0 + D*E_0	I0 + D*E_1
F	F*C_0	F*C_1	F*G	I_0	I_0
$\mathbf{C}_0$*$\mathbf{E}_0$	0	-D*E_0	C_0×E_0*G	I_0	I_0 + F*C_0
$\mathbf{C}_0$*$\mathbf{E}_1$	0	-D*E_1	C_0×E_1*G	I_0 - F*C_0	I_0
$\mathbf{C}_0$*$\mathbf{E}_2$	0	-D*E_2	C_0×E_2*G	I_0 - F*C_0	I_1
$\mathbf{C}_1$*$\mathbf{E}_0$	D*E_0	0	C_1×E_0*G	I_0	I_0 + F*C_1
$\mathbf{C}_1$*$\mathbf{E}_1$	D*E_1	0	C_1×E_1*G	I_0 - F*C_1	I_0
$\mathbf{C}_1$*$\mathbf{E}_2$	D*E_2	0	C_1×E_2*G	I_0 - F*C_1	I_1
$\mathbf{C}_2$*$\mathbf{E}_0$	D*E_0	H_1	C_2×E_0*G	I_0	I_0 + F*C_2
$\mathbf{C}_2$*$\mathbf{E}_1$	D*E_1	H_1	C_2×E_1*G	I_0 - F*C_2	I_0
$\mathbf{C}_2$*$\mathbf{E}_2$	D*E_2	H_1	C_2×E_2*G	I_0 - F*C_2	I_1

3.1. táblázat. A számítások táblázata.

3.1. Függvénykönyvtár tervezése

A függvénykönyvtár C nyelven [5] íródott C11 szabványban. Ez egy egyszerűen megtanulható, gyors programozási nyelv. Létrehozhatunk benne függvényeket, új típusokat/-struktúrákat, amelyet később bárhol használhatunk a program készítése során. A készített függvénykönyvtár könnyen használható, mindössze bele kell raknunk az általunk készített program forráskódjai közé a collision_triangle.c, illetve collision_triangle.h fájlokat, majd beimportálni azt a forráskódban a '#include "collision_triangle.h"' sor segítségével. Ha mindent helyesen csináltunk, akkor máris használhatóak a függvények, illetve struktúrák, amelyeket a 3.2, illetve 3.3 táblázat mutatja be.

Változó	**Típusa**	**Leírása**
x	Float	Egy háromdimenziós pont helyét jelöli az x tengelyen.
y	Float	Egy háromdimenziós pont helyét jelöli az y tengelyen.
z	Float	Egy háromdimenziós pont helyét jelöli az z tengelyen.

3.2. táblázat. Vec3 struktúra bemutatása

Függvény	Visszatérése	Paraméterei	Leírása
check_collision	Boolean	modell1 modell2 modell1 pozíció modell2 pozíció	A modellek és a modellek pozícióinak megadásával visszakapjuk, hogy ütköznek-e vagy sem.
sub	Vec3	2 térbeli pont (A,B)	Visszaadja a két pont különbségét.
cross	Vec3	2 térbeli pont (A,B)	Visszaadja a két pont Descartes szorzatát.
dot	Float	2 térbeli pont (A,B)	Visszaadja a 2 pont koordinátáinak szorzatát (A.x * B.x + A.y * B.y + A.z * B.z)
getmin	Float	3 float változó (a,b,c)	Visszaadja a 3 változó közül a legkisebbet.
getmax	Float	3 float változó (a,b,c)	Visszaadja a 3 változó közül a legnagyobbat.
get_distance	Float	2 térbeli pont (A,B)	Visszaadja a 2 pont közötti távolságot.
scale_model	Void	modell 3 float változó a méretezéshez	Átméretezi az adott modellt.
rotate_modele	Void	modell 3 float változó a forgatáshoz	Forgatja a térben a modellt.
mirror_model	Void	modell tengely indexe 0=z,1=y,2=x	Tükrözi a modellt a megadott tengelyre.

3.3. táblázat. Függvények bemutatása

4. fejezet

Megvalósítás

Háromszöget metszéspontjának számításához több, a **C** nyelvbe alapértelmezetten be nem épített, szinte már elemi szintű függvényre van szükségünk.
Ilyen például a minimum, illetve maximum kiválasztása 3 float-ból a math.h függvénykönyvtár segítségével:

```
float getmin(float a, float b, float c)
{
    return fminf(fminf(a, b), c);
}

float getmax(float a, float b, float c)
{
    return fmaxf(fmaxf(a, b), c);
}
```

Illetve az elemi szintű számítások, például 3 dimenziós vektorok kivonása, szorzása, Descartes szorzása:

```
vec3 sub(vec3 A, vec3 B)
{
    vec3 C;
    C.x = A.x - B.x;
    C.y = A.y - B.y;
    C.z = A.z - B.z;
    return C;
}

float dot(vec3 A, vec3 B)
{
    return A.x * B.x + A.y * B.y + A.z * B.z;
}
```

```
vec3 cross(vec3 A, vec3 B)
{
    vec3 C;
    C.x = A.y * B.z - A.z * B.y;
    C.y = -(A.x * B.z - A.z * B.x);
    C.z = A.x * B.y - A.y * B.x;
    return C;
}
```

Illetve a korábban említett eldöntés, hogy a két háromszög metszi-e egymást, vagy sem.

```
bool isSeparated(float a1, float a2, float b0, float b1, float b2)
{
    float a0 = 0;
    if (fmaxf(getmin(a0, a1, a2), getmax(b0, b1, b2))
    == getmin(a0, a1, a2))
    {
        return true;
    }
    if (fmaxf(getmax(a0, a1, a2), getmin(b0, b1, b2))
    == getmin(b0, b1, b2))
    {
        return true;
    }
    return false;
}
```

4.1. Metszéspont számítása

Az előző szekcióban bemutatott számítások felhasználásával mostmár kiszámíthatjuk, hogy a háromszögek metszik-e egymást, vagy sem. Erre a check_collision függvényt használjuk.

```
bool check_collision(Model *model1, Model *model2, vec3 model1_position,
vec3 model2_position)
{
    int k = 0, l = 0;
    float limit, distance;
    limit = model1->farestpoint + model2->farestpoint + 0.5;
    distance = get_distance(model1_position, model2_position);
    if (fminf(distance, limit) == limit)
    {
        return false;
    }
    while (k < model1->i_f)
    {
        while (l < model2->i_f)
        {
            if (check_collision_helper(k, l, model1, model2,
            model1_position, model2_position))
            {
                return true;
            }
            l++;
        }
        k++;
        l = 0;
    }
    return false;
}
```

A függvénynek 4 bemenete van. Az első 2 bemenet adja meg, hogy melyik 2 modellt szeretnék leellenőrizni, hogy ütköznek-e. Az utolsó 2 bemenet adja meg a modellek jelenlegi pozícióját. Elsőként leellenőrizzük a két modell közötti távolságot. Ha kiesnek egymás távolságából, akkor nem ellenőrizzük le a metszéseket, ezzel optimalizálva a programot.

Amennyiben közel vannak egymáshoz a modellek, akkor egy dupla ciklus segítségével végigmegyünk mindkét modell minden egyes háromszögén és leellenőrizzük a check_collision_helper függvény segítségével, hogy ütköznek-e. Ha ütköznek, akkor igaz értékkel visszatérünk, és nem ellenőrzünk tovább.

Az egyszerűbb olvashatóság kedvéért egy check_collision_helper segédfüggvényt használunk a metszéspontok számításához.

```
bool check_collision_helper(int k, int l, Model *model1,
Model *model2, vec3 model1_position, vec3 model2_position)
{
    vec3 A0, A1, A2, B0, B1, B2;

    A0.x = model1->v[model1->f[k].points[0].vertex_index].x
    + model1_position.x;
    A0.y = model1->v[model1->f[k].points[0].vertex_index].y
    + model1_position.y;
    A0.z = model1->v[model1->f[k].points[0].vertex_index].z
    + model1_position.z;
    A1.x = model1->v[model1->f[k].points[1].vertex_index].x
    + model1_position.x;
    A1.y = model1->v[model1->f[k].points[1].vertex_index].y
    + model1_position.y;
    A1.z = model1->v[model1->f[k].points[1].vertex_index].z
    + model1_position.z;
    A2.x = model1->v[model1->f[k].points[2].vertex_index].x
    + model1_position.x;
    A2.y = model1->v[model1->f[k].points[2].vertex_index].y
    + model1_position.y;
    A2.z = model1->v[model1->f[k].points[2].vertex_index].z
    + model1_position.z;

    return checkCollisionTriangle(A0, A1, A2, B0, B1, B2);
}
```

6 bemenete van a függvénynek. Az első bemenet határozza meg, hogy az adott modell hanyadik háromszögét vizsgáljuk. A második bemenet határozza meg, hogy a másik modell hanyadik háromszögét vizsgáljuk. Utána következik a 2 modell, majd a modellek pozíciója.

Létrehozunk 6 darab 3 dimenziós vektort, amelyek tartalmazzák majd a háromszöget csúcspontjainak koordinátáit a térben.

Kiszámításának módja: model1->f[k].points[0].vertex_index model1 az adott modell, amely egy pointerként rámutat az f struktúrára, amely tartalmazza az adott háromszög adatait. Az f k-adik indexű háromszögét vizsgáljuk mindig, azon belül a points struktúrában tárolt adatokra van szükségünk. Ennek a változónak is a vertex_index elemére. Ezzel megkapjuk egy háromszög pontjának az indexét.

Ebből kinyerhetjük a háromszög csúcspontjainak pontos koordinátáit.

A0.x = model1->v[model1->f[k].points[0].vertex_index].x + model1_position.x
Így beállíthatjuk az új háromszögünk első csúcspontjának X koordinátáját, amelyhez hozzá kell adnunk a modell jelenlegi pozícióját. Ezt megismételjük mindig csúcspontra, illetve a másik modellre is, mielőtt meghívjuk a checkCollisionTriangle függvényt.

A checkCollisionTriangle függvény megkapja az előző függvény által kiszámolt 6 csúcspontot. Ezen csúcspontok segítségével számolja ki a fentebb említett számolási módszer segítségével az éleket, a normál vektorokat és a távolságot, majd leellenőrzi, hogy a háromszögek metszik-e egymást, vagy sem.

```
bool checkCollisionTriangle(vec3 A0, vec3 A1, vec3 A2, vec3 B0,
vec3 B1, vec3 B2)
{
    vec3 C0 = sub(A1, A0);
    vec3 C1 = sub(A2, A0);
    vec3 C2 = sub(C1, C0);
    vec3 D = cross(C0, C1);
    vec3 E0 = sub(B1, B0);
    vec3 E1 = sub(B2, B0);
    vec3 E2 = sub(E1, E0);
    vec3 F = cross(E0, E1);
    vec3 G = sub(B0, A0);
    ...
```

A metszések kiszámítása kódban terjedelmes, ezért csak 1 példa lesz bemutatva.

```
float h1, i0, i1, i2;

// Separate: D
i0 = dot(D, G);
i1 = i0 + dot(D, E0);
i2 = i0 + dot(D, E1);
if (isSeparated(0, 0, i0, i1, i2))
{
    return false;
}
```

Itt a D normálvektor alapján próbáljuk eldönteni, hogy különállnak-e a háromszögek, vagy sem. A 3.1 táblázat alapján kiszámítjuk a H_1, H_2, I_0, I_1, I_2 változókat, majd az alapján meghívjuk az isSeparated függvényt, amely eldönti, hogy a háromszögek különállnak-e, vagy sem. A könnyebb átláthatóság érdekében a függvények kapcsolatát a 4.5 ábra mutatja.

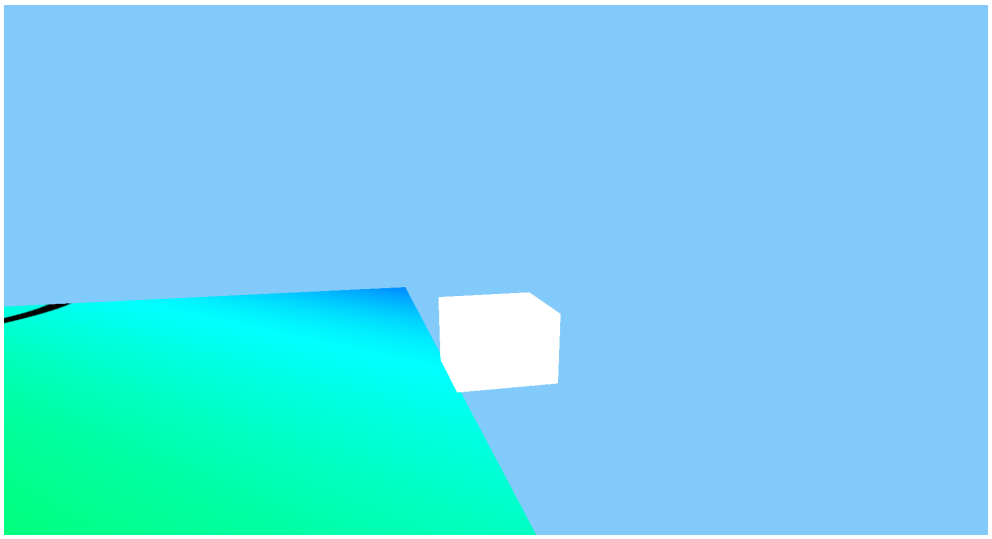
4.2. Modell mozgatása a térben

Az eddig bemutatott kódok alapján a program így csak az el nem forgatott, át nem méretezett modelleket képes kezelni, ami nem kedvező számunkra. Ezért új függvények lettek létrehozva a modellek méretezéséhez, forgatásához, tükrözéséhez. Ha azt szeretnénk, hogy a modell "hitboxa"[1.1] mozogjon a modellel együtt, akkor ezeket a függvényeket kell használnunk az OpenGL[3]-ben alapértelmezett függvények helyett. Ezek a függvények magát a modellt módosítják, nem pedig a megjelenítését. Az egyszerűbb átláthatóság érdekében készült egy demo program, amely segítségével vizuálisan is láthatjuk a függvények működését. Az ebben a fejezetben használt képek a demo program segítségével lettek elkészítve, amelyben a felhasználó körbe tudja járni a teret, különböző modelleket létrehozva, és letesztelni az azokkal való ütközést. A program használatát és billentyűzetkiosztását a 4.4 fejezet mutatja be részletesen.

4.2.1. Modell méretezése

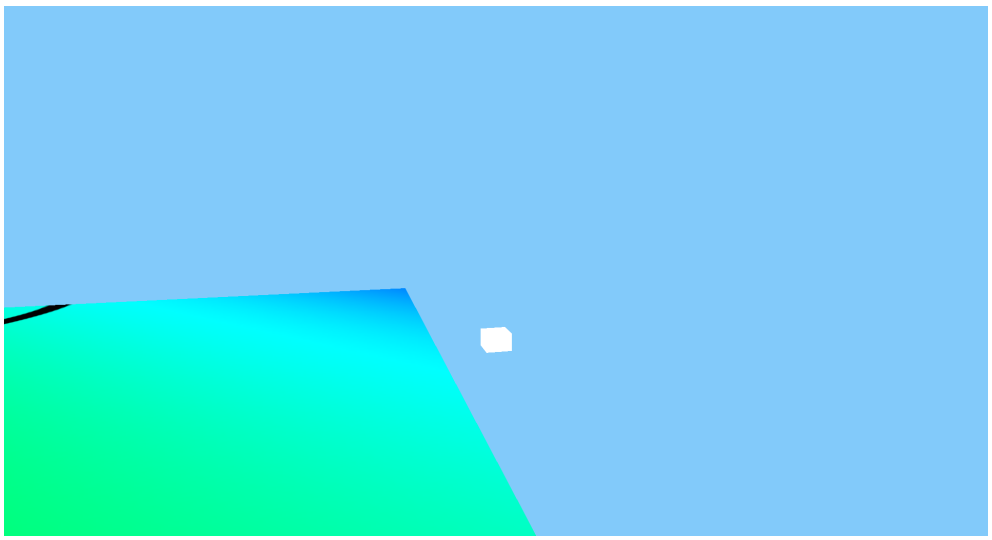
A scale_model függvény 4 bemenettel rendelkezik, az első a modell maga, a maradék 3 pedig a méretezéshez szükséges float változók. A függvény végigmegy minden egyes háromszögén a modellnek, és a háromszög csúcspontjainak a koordinátáit megszorozza a megadott új méretekkel. Modellek méretezéséről példát a 4.1, illetve 4.2 ábrákon láthatunk.

```
void scale_model(Model *model, float scalex, float scaley,
float scalez)
{
    int k = 0;
    while (k < model->i_f)
    {
        for (int i = 0; i < 3; i++)
        {
            model->v[k].x *= scalex;
            model->v[k].y *= scaley;
            model->v[k].z *= scalez;
        }
        k++;
    }
}
```



4.1. ábra. Alapértelmezett playermodel[5.1] méret

⇓

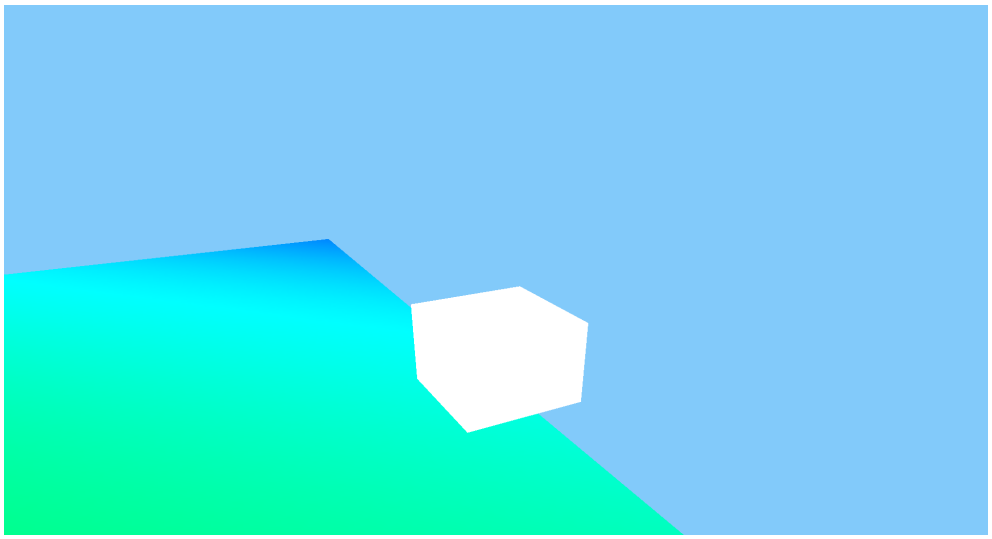


4.2. ábra. Playermodel[5.1] méretének csökkentése

4.2.2. Modell forgatása

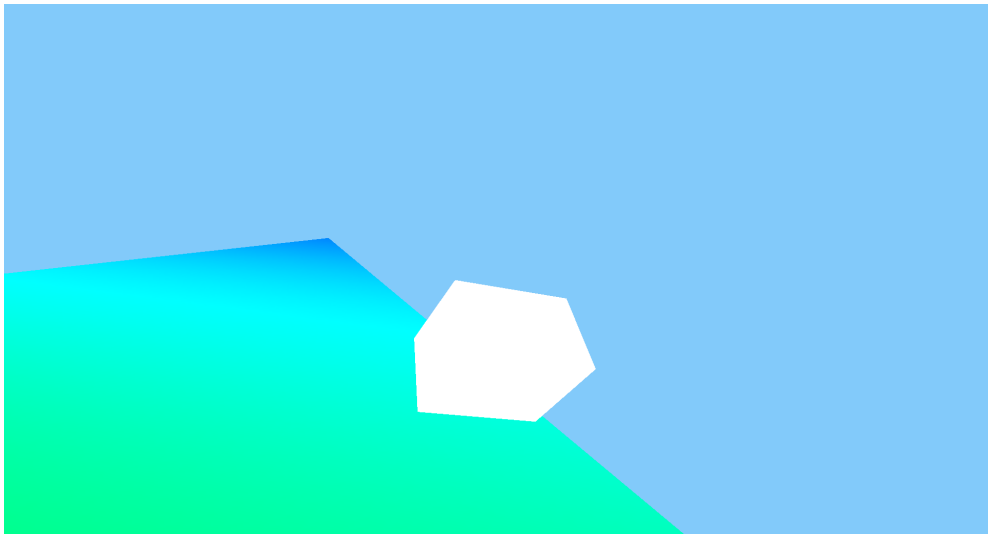
A modell forgatása az egyik legnehezebb feladat az összes közül. Itt nem elég szimplán szoroznunk, hanem a forgatás miatt koszinusz, illetve szinusz számításokat is végeznünk kell. A függvény végigmegy minden egyes háromszögén a modellnek, majd a csúcspontok koordinátáit módosítja szinusz és koszinusz számítással az adott forgatási szögek alapján. Modellek forgatásáról példát a 4.3, illetve 4.4 ábrákon láthatunk.

```
void rotate_model(Model *model, float anglex, float angley, float anglez)
{
    int k = 0;
    float cord1, cord2, angle;
    while (k < model->i_f)
    {
        cord1 = model->v[k].x;
        cord2 = model->v[k].y;
        angle = anglex * (float)(M_PI / 180);
        model->v[k].x = cord1 * cosf(angle) - cord2 * sinf(angle);
        model->v[k].y = cord1 * sinf(angle) + cord2 * cosf(angle);

        cord1 = model->v[k].x;
        cord2 = model->v[k].z;
        angle = angley * (float)(M_PI / 180);
        model->v[k].x = cord1 * cosf(angle) - cord2 * sinf(angle);
        model->v[k].z = cord1 * sinf(angle) + cord2 * cosf(angle);

        cord1 = model->v[k].y;
        cord2 = model->v[k].z;
        angle = anglez * (float)(M_PI / 180);
        model->v[k].y = cord1 * cosf(angle) - cord2 * sinf(angle);
        model->v[k].z = cord1 * sinf(angle) + cord2 * cosf(angle);
        k++;
    }
}
```



4.3. ábra. Alapértelmezett playermodel[5.1]

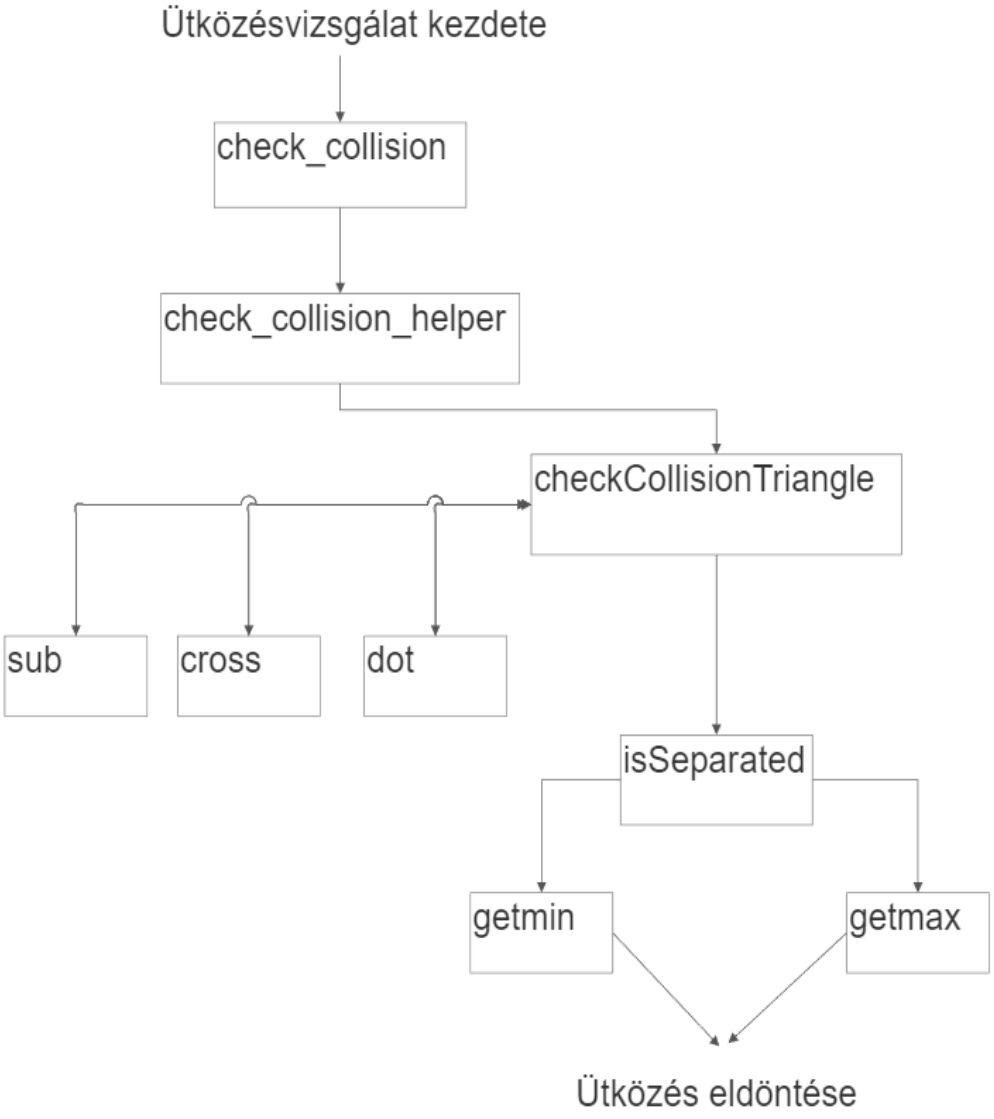
⇓



4.4. ábra. Playermodel[5.1] forgatása a térben

4.3. Függvények kapcsolata

A 4.5. hívási fa bemutatja a program fő függvényeit és azok közötti kapcsolatokat. A check_collision a kezdőpont, ahonnan indul maga az ütközésvizsgálat, ezt az egyetlen függvényt kell meghívnunk, ha 2 modellről el szeretnénk dönteni, hogy metszi-e egymást, vagy sem. Maga a modellek ütközésének vizsgálata a checkCollisionTriangle csomópontnál történik a sub, cross és dot függvények segítségével, majd az isSeparated eldönti a hívás kimenetelét, amely true, vagy false érték lehet. True érték esetén a modellek ütköznek, false érték esetén nem ütköznek.



4.5. ábra. Függvénykönyvtár hívási fája

4.4. Demo program működtetése

A függvénykönyvtár teszteléséhez készítettem egy programot, amellyel szemléltethetjük a számításokat, a számítások működését és azok hatékonyságát. A teszt programban egy úgynevezett "playermodel"[5.1]-t irányíthatunk, amely egy egyszerű kocka modell. A playermodellel[5.1] körbejárhatjuk a teret, a modellt forgathatjuk, tükrözhetjük, méretezhetjük, illetve más modellekkel ütközhetünk. A program működtetéséhez használható különböző billentyűkombinációkat a 4.1 táblázat szemlélteti.

Billentyű	**Esemény**
W,A,S,D	A "playermodel"[5.1] mozgatása a térben X,Y tengelyen.
CTRL, SPACE	A "playermodel"[5.1] mozgatása a térben Z tengelyen.
Egér	A kamera mozgatása a "playermodel"[5.1] körül.
Görgő	A kamera távolságának állítása a "playermodelhez"[5.1] képest.
E	Új modell létrehozása az ütközésvizsgálat teszteléséhez.
Q	A "playermodel"[5.1], illetve a "hitbox"[1.1] ki/be kapcsolása.
ESC	Kilépés a programból.
N, M	A "playermodel"[5.1] méretének növelése/csökkentése.
C, V, B	A "playermodel"[5.1] tükrözése X, Y és Z tengelyekre.
J, L	A "playermodel"[5.1] forgatása az X tengelyen.
I, K	A "playermodel"[5.1] forgatása az Y tengelyen.
U, O	A "playermodel"[5.1] forgatása az Z tengelyen.

4.1. táblázat. Demo program használata

5. fejezet

Optimalizálás

Minél több a "playermodel"[5.1] háromszögeinek száma, illetve a többi modellek száma, és azok háromszögeinek száma, annál több számítást kell végeznünk. Például 2 modell esetén, ha a playermodellünk[5.1] 100 háromszögből áll, és a másik modellünk is szintén 100 háromszögből áll, akkor a programnak frame-enként 100·100, azaz 10 000 számítást kell végeznie, négyzetes idejű az algoritmus számítási ideje. Emiatt a módszer nem alkalmas valós idejű alkalmazásokhoz. Több módszert is kipróbáltam lehetséges optimalizálás szempontjából.

5.1. Szűrés távolság alapján

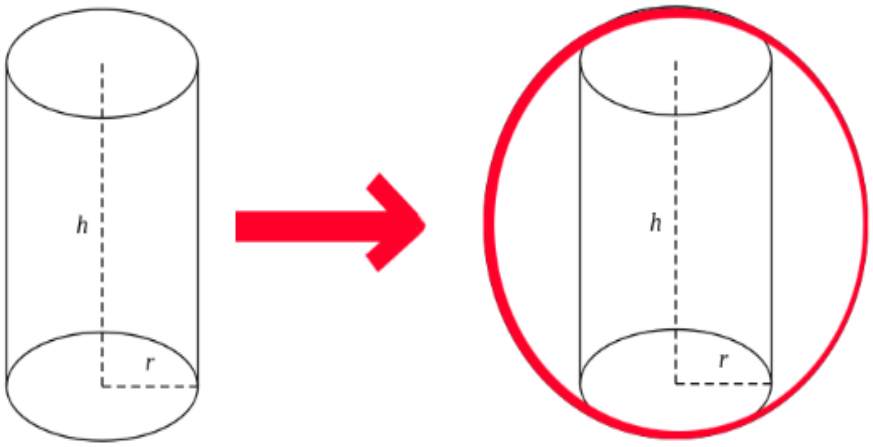
Minden modell betöltésekkor kiszámítjuk a modell középpontjától számított legtávolabbi pontjának hosszát, ahogy a 5.1 ábrán láthatjuk. Ezt csak egyszer kell kiszámítanunk, kivéve ha futtatás közben szeretnénk módosítani a modellt, például a modell méretének megváltoztatásával. Ütközésvizsgálat előtt kiszámítjuk, hogy a két modell ütközhet-e egymással vagy sem. Ez lényegében a négyzetes elválasztás(2.1.1) módszere alapján működik.
Részlet a check_collision függvényből:

```
float limit, distance;
limit = model1->farestpoint + model2->farestpoint + 0.5;
distance = get_distance(model1_position, model2_position);
if (fminf(distance, limit) == limit)
{
    return false;
}
```

5.1. definíció (Playermodel). A "playermodel" egy számítógépes játékban használt grafikus megjelenítése a játékos karakternek, amely magában foglalja annak kinézetét és animációit a játék világában való mozgása során.

A limit változóban megnézzük a maximális távolságot, amelyen belül már ütközésvizsgálatot vizsgálunk. Összeadjuk a két modell legtávolabbi pontjának távolságát, illetve egy extra konstanst. Ezután a distance változóba elmentjük a két modell közötti távolságot. Ha a limit kisebb, mint a distance, akkor nem lehetséges a két modell ütközése, azért visszatérünk false értékkel, nem vizsgálunk ütközést.

Az optimalizálás ezen esetben **sikeres**, kevesebb számítást veszünk figyelembe, a program gyorsul, amely függ a modellek távolságától, méretétől és komplexitásától.



5.1. ábra. Modell köré egy gömb alakú "hitbox"[1.1] generálása

5.2. Távolság szűrése háromszögekkel

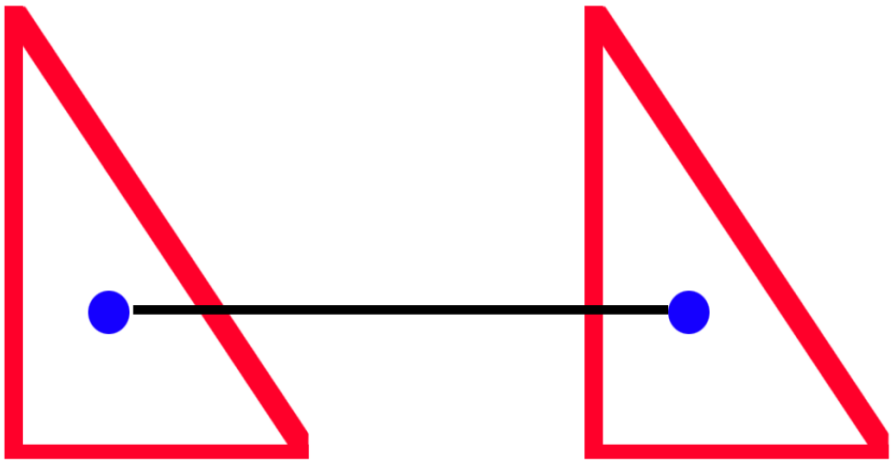
Lényege, hogy ütközésvizsgálat előtt a modellek háromszögeinek a középpontját kiszámítjuk, majd a középpontokat viszonyítjuk egymáshoz, ahogy a 5.2 ábrán láthatjuk. Ha a háromszögek közötti távolság nagyobb, mint a maximum táv, akkor az adott háromszög számítása elhanyagolható, mivel biztosra vehetjük, hogy nem lesz az adott két háromszög esetén metszés.

Az optimalizálás ezen esetben sikertelen, a számítások ideje drasztikusan megnő, használata nem ajánlott.

5.3. Távolság szűrése háromszögek mentésével

Lényege ugyan az, mint az előző szekcióban bemutatott algoritmusnak, annyi kivétellel, hogy a középpontokat csak a modell beolvasásakor számítjuk ki, majd lementjük későbbi használatra, ahogy a 5.2 ábrán láthatjuk.

Az optimalizálás ezen esetben sikertelen, a számítások ideje megnő az optimalizálatlan programhoz képest, de csökken az 5.2 algoritmushoz képest, illetve 25%-al megnő a memóriaigény is. Használata nem ajánlott.

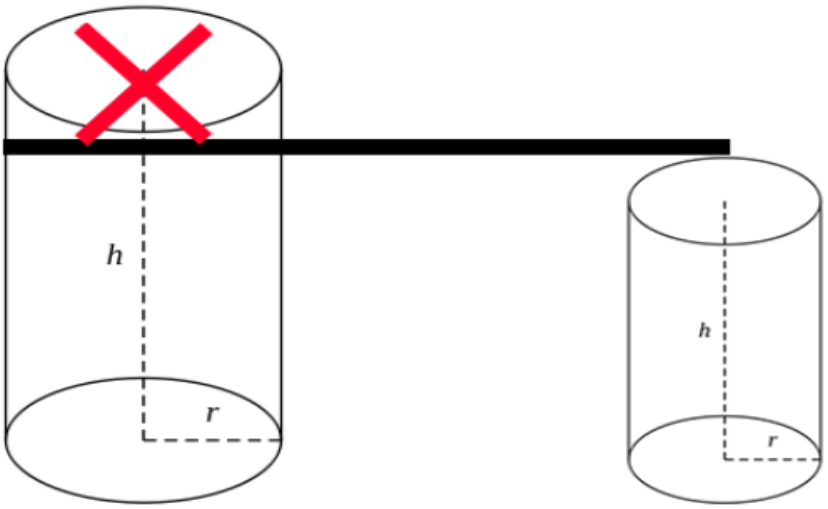


5.2. ábra. Távolság szűrése háromszögek középpontjával

5.4. Pozíció alapján szűrés

Lényege, hogy az adott modell nem feltétlenül van mindig ugyan abban a síkban, mint a másik vizsgálandó modell. Megnézzük a két modell középpontjának koordinátáit, illetve a legtávolabbi pontját az adott tengelyhez képest, ahogy az 5.3 ábrán láthatjuk. Például, ha a playermodellünk a Z koordinátán 20 pixel magas, akkor a maximum szint a Z tengelyen az a modell középpontja + 20 pixel lesz, minden más háromszöget az adott modellben figyelmen kívül hagyunk. Ezeket a lépéseket meg kell ismételnünk minden tengelyre 2 alkalommal. Ez összesen 6 különböző számítást jelent.

Az optimalizálás ezen esetben megoldható, a számítások ideje csökken, de cserébe a memória igény nő.



5.3. ábra. Tengelyek alapján szűrés

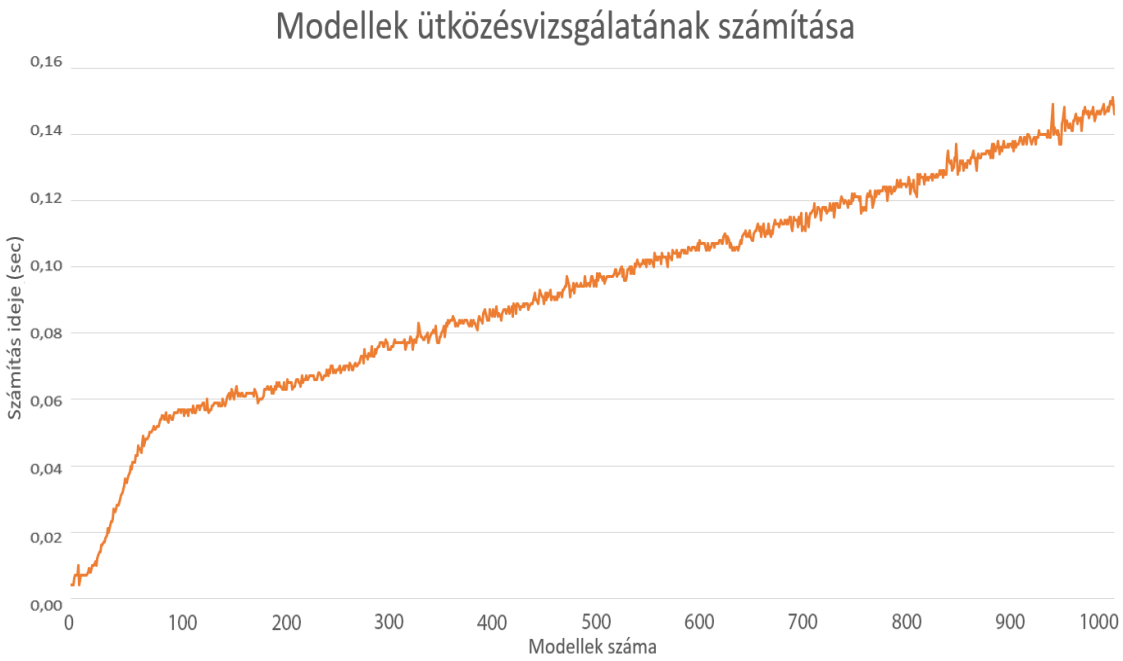
6. fejezet

Számítási idő

A függvénykönyvtár, illetve a demo program futási idejét 2 különböző módon tesztelhetjük. Első esetben csak a háromszögek metszéspontjának számítása lett figyelembe véve, míg a második esetben az OpenGL [3] renderelési ideje is. A számítások intel core i9-13900 processzoron, illetve RTX 4070 videókártyán lettek elvégezve. Az eredmények más rendszereken mások lehetnek. Modellek ütközésének vizsgálatakor mindig 1 kockát (12 háromszög) és több hengert (124 háromszög * modellek száma) vettünk figyelembe.

6.1. ábra. Háromszögek metszéspontjának számítása

Háromszögek metszéspontjainak számítási idejét láthatjuk a 6.1 ábrán, ahogy a számítási idő lineárisan nő a háromszögek száma alapján.



6.2. ábra. Modellek ütközésvizsgálatának számítása

Modellek ütközésvizsgálatánál már nem mondhatjuk el, hogy teljesen lineáris lenne a számítási igény. Az elején a 6.2 ábrán (kb. 100 modellig) hirtelen nő a számítási idő, majd egyre inkább egyenletes lesz, lineárisan nő a szükséges idő.

7. fejezet

Összefoglalás

A szakdolgozat eredménye egy könnyen használható függvénykönyvtár, amely lehetővé teszi 3 dimenziós modellek beimportálását, valamint "hitbox"[1.1]-ként kezelését, ütközések vizsgálatát más modellekkel. A program fejlesztése során kiemelt figyelmet kapott, hogy az alkalmazást könnyen használhassák más fejlesztők, akik esetleg nem jártasak, vagy csak kevés tapasztalattal rendelkeznek ütközések vizsgálatában.

A beépített funkciók azonnal elérhetővé válnak, amint beimportáljuk a függvénykönyvtárat a fejlesztői környezetbe. Fontos megemlítenünk, hogy a függvénykönyvtár használatához a fejlesztőnek szüksége van egy C fordítóra, illetve rendelkeznie kell OpenGL [3] és SDL2 [4] függvénykönyvtárakkal is. A fejlesztő döntheti el, hogy mikor és milyen modellekre alkalmazza a vizsgálatot, illetve milyen eseményekkel jár az ütközés, ezzel személyre szabható a program minden része.

Az ütközésvizsgálat egyszerűen elvégezhető a check_collision függvény meghívásával, illetve a szükséges adatok megadásával. A függvény eredményeként visszatér egy igaz, vagy hamis értékkel, ezzel jelezve, hogy a modellek ütköztek-e egymással, vagy sem.

A jövőben számos további fejlesztés valósulhat meg a program szempontjából, ilyen lehet például az optimalizálás, amellyel a teljesítményt növelhetjük, míg az erőforrások nagyságát csökkenthetjük, valamint még könnyebbé, felhasználóbarátabbá tehetjük a program használatát.

A függvénykönyvtár széles körben alkalmazható fizikai szimulációk, videójátékok esetében, ezzel hozzájárulva a fejlesztők munkájához, illetve a számítógépes grafikában rejlő lehetőségek kiaknázásához.

Irodalomjegyzék

[1] David Eberly. Dynamic collision detection using oriented bounding boxes. *Journal of Geometric Tools*, 4(1):23–24, 1999.

[2] Christer Ericson. Real-time collision detection. *Comprehensive guide to the components of efficient real-time collision detection systems*, 2004.

[3] Silicon Graphics. Open graphics library (opengl). *Cross-platform application programming interface*, 1992.

[4] Sam Lantinga. Simple directmedia layer 2.0 (sdl2). *Cross-platform software development library*, 1998.

[5] Dennis Ritchie. The c programming language. *General-purpose computer programming language*, 1978.

CD Használati útmutató

A függvénykönyvtár használatához szükségünk van egy OpenGL[3] alapú programra. Amennyiben nem rendelkezünk ilyennel, a szakdolgozat tartalmaz egy demo programot, amelyen tesztelhetjük a funkciókat.

A demo elindításához nem kell mást tennünk, mint elindítani a start.bat file-t a program nevezetű mappában.

A CD lemez tartalmazza:

- a dolgozatot egy `dolgozat.pdf` fájl formájában,
- a LaTeX forráskódját a dolgozatnak a szakdolgozat nevezetű mappában,
- az elkészített programot a program nevezetű mappában,
- a használt matematikai képletet tartalmazó pdf-et DynamicCollisionDetection néven,
- egy útmutatót a CD használatához README.md néven.