Szakdolgozat

Miskolci Egyetem

Ütközésvizsgálat automatikus optimalizálása térbeli modellekhez

Készítette:

Szöllősi János

Programtervező informatikus

Témavezető:

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: 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 olyan függvénykönyvtár létrehozása, amely automatikusan létrehoz egy beimportált 3 dimenziós modellhez egy "hitboxot", amely alapján kiszámíthatjuk az adott modell ütközését más modellekkel. A program **C** nyelvben íródott, az **OpenGL** és **SDL2** függvénykönyvtárak segítségével. A program fő célja a modellek felbontása atomi szintre (háromszögekre), majd a háromszögek metszéspontjának kiszámítása.

A függvénykönyvtár tökéletes lehet kisebb játékok fejlesztéséhez, vagy egyéb fizikai szimulációk motorjaként. Mivel az ütközésvizsgálat atomi szinten történik a modell háromszögekre bontásával, így nagy az ütközésvizsgálat pontossága.

2. fejezet

Koncepció

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

A valóságban az emberi gondolkodásnak egyszerűnek tűnhet eldönteni, hogy két háromszög metszi-e egymást, vagy sem. Programozás, illetve matematika terén 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 [1](4.1 Separation of Triangles) szekciót találtam. A dokumentáció tökéletesen elmagyarázza a matematikai képletek elemeit, azok használatát, lépéseit. Ezek mind táblázatba szedve találhatóak. A pontos magyarázat a következő fejezetben található. Emellett rengeteg különböző módszer található az interneten térbeli modellek ütközésének vizsgálatához, például:

•**Négyzetes elválasztás:** Lényege, hogy ha két test nem ütközik egymással, akkor a két test között áthaladva létezik egy olyan sík, amely nem metszi őket. Konvex modellek esetén hatékony.

•**Sugárkövetés:** Lényege, hogy minden modellhez egy halmazt rendelnek, amely tartalmazza a modellt felépítő pontokat. Az átfedő pontokkal vizsgálhatjuk az ütközést.

•**Voxel-alapú:** Lényege, hogy a térbeli modellt voxelek halmazaként írjuk le. Az ütközést a voxelek átfedésével vizsgálhatjuk. Bonyolult formájú modellek esetén hatékony.

•**Egyenletek alkalmazása:** Lényege, hogy néhány modell (például: gömbök, síkok, hengerek) már tartalmazza az ütközésvizsgálathoz szükséges információkat. Ilyen modellek esetén egyenleteket használunk az ütközésvizsgálat detektálására.

3. fejezet

Tervezés

A függvénykönyvtár teszteléséhez készült egy program, amellyel szemléltethetjük a számításokat, a számítások működését és azok hatékonyságát. A teszt programban egy úgynevezett "playermodell"-t irányíthatunk, amely egy egyszerű kocka modell. A playermodellel körbejárhatjuk a teret, a modellt forgathatjuk, tükrözhetjük, méretezhetjük, illetve más modellekkel ütközhetünk. Ezen lépésekhez különböző billentyűkombinációk érhetőek el.

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

3.1. Számítások

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

$$\mathbf{D} = \mathbf{C}_0 \times \mathbf{C}_1$$

A számítások megkönnyítéséhez kiszámítjuk az eltolásvektort is.

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

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

$$\mathbf{H}_1 = \mathbf{L} * \mathbf{C}_0$$

$$\mathbf{H}_2 = \mathbf{L} * \mathbf{C}_1$$

$$\mathbf{I}_0 = \mathbf{L} * \mathbf{G}$$

$$\mathbf{I}_1 = \mathbf{I}_0 + \mathbf{L} * \mathbf{E}_0$$

$$\mathbf{I}_2 = \mathbf{I}_0 + \mathbf{L} * \mathbf{E}_1$$

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

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

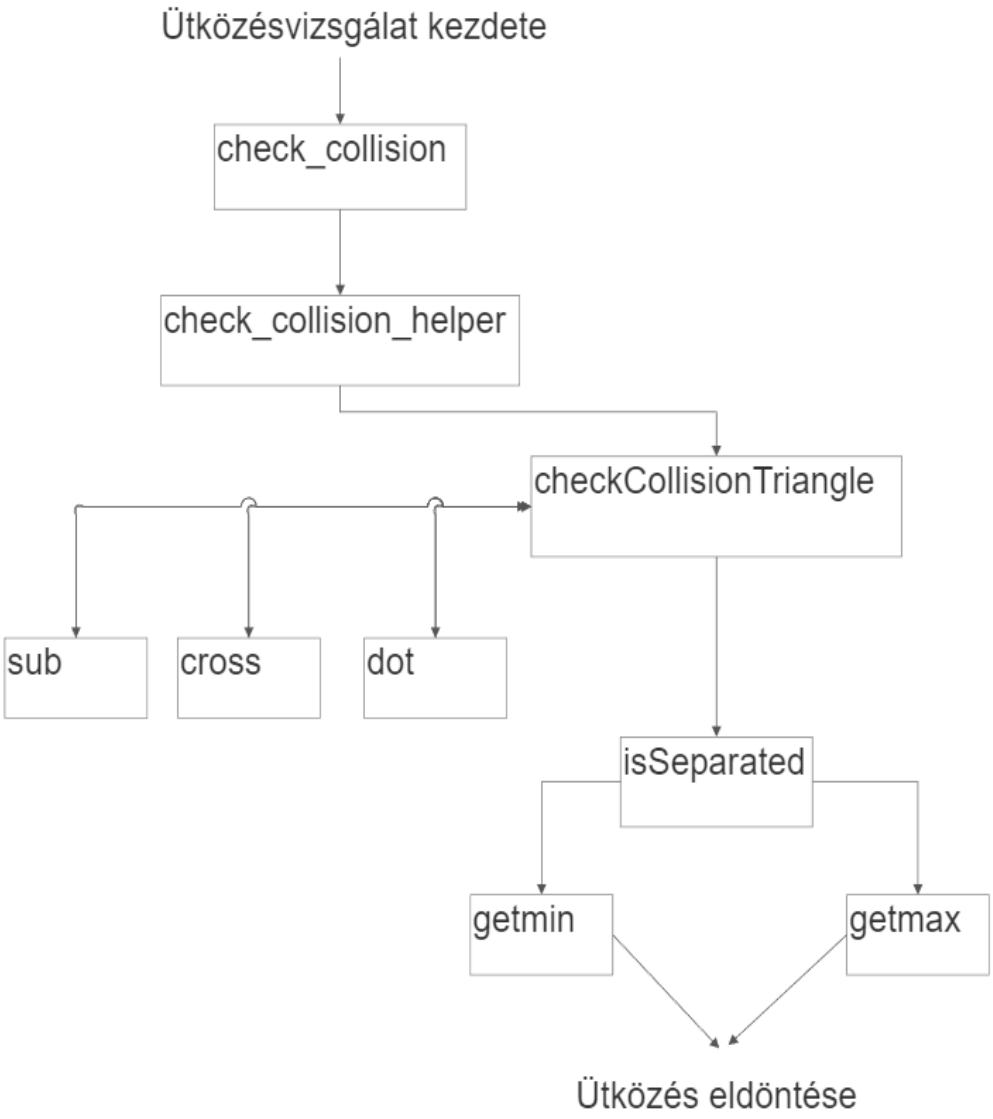
, akkor a két háromszögre biztosan mondható, hogy nem metszik egymást. Ez esetben nem szükséges további számításokat végezni. Amennyiben nemleges választ kapunk, akkor tovább kell számítanunk minden sort. Ha az utolsó sor esetén sem kapunk pozitív választ, akkor kimondhatjuk, hogy a két háromszög metszi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

3.1. táblázat. A számítások táblázata.

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_0xE_0*G	I_0	I_0 + F*C_0
$\mathbf{C}_0$*$\mathbf{E}_1$	0	-D*E_1	C_0xE_1*G	I_0 - F*C_0	I_0
$\mathbf{C}_0$*$\mathbf{E}_2$	0	-D*E_2	C_0xE_2*G	I_0 - F*C_0	I_1
$\mathbf{C}_1$*$\mathbf{E}_0$	D*E_0	0	C_1xE_0*G	I_0	I_0 + F*C_1
$\mathbf{C}_1$*$\mathbf{E}_1$	D*E_1	0	C_1xE_1*G	I_0 - F*C_1	I_0
$\mathbf{C}_1$*$\mathbf{E}_2$	D*E_2	0	C_1xE_2*G	I_0 - F*C_1	I_1
$\mathbf{C}_2$*$\mathbf{E}_0$	D*E_0	H_1	C_2xE_0*G	I_0	I_0 + F*C_2
$\mathbf{C}_2$*$\mathbf{E}_1$	D*E_1	H_1	C_2xE_1*G	I_0 - F*C_2	I_0
$\mathbf{C}_2$*$\mathbf{E}_2$	D*E_2	H_1	C_2xE_2*G	I_0 - F*C_2	I_1

4. fejezet

Megvalósítás



4.1. ábra. Függvénykönyvtár blokk diagramja

4.1. Elemi szintű számítások

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;
}
```

4.2. 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 valamennyire a programot.

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

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 háromszöget 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 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. (Az *isSeparated* függvény megtalálható az "Elemi szintű számítások" szekcióban)

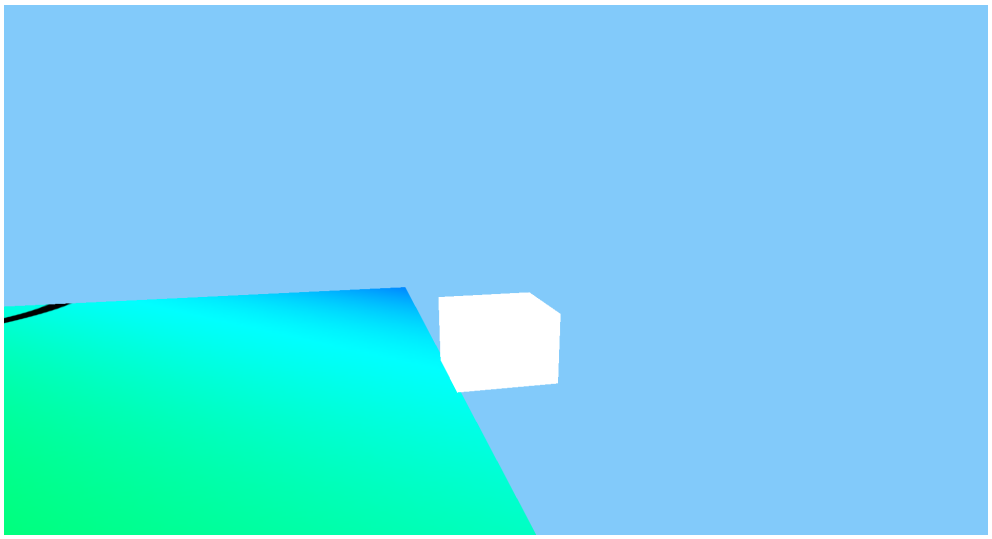
4.3. Modell formázása

Az eddig bemutatott kódok alapján a program így csak az alapértelmezett modelleket képes kezelni, ami nem kedvező számunkra. Ezért új függvények lettek létrehozva a modellek méretezéséhez, forgatásához, tükrözéséhez. Ha azt szeretnénk, hogy a modell "hitboxa" mozogjon a modellel együtt, akkor ezeket a függvényeket kell használnunk az **OpenGL**-ben alapértelmezett függvények helyett. Ezek a függvények magát a modellt módosítják, nem pedig a megjelenítését.

4.3.1. Modell méretezése

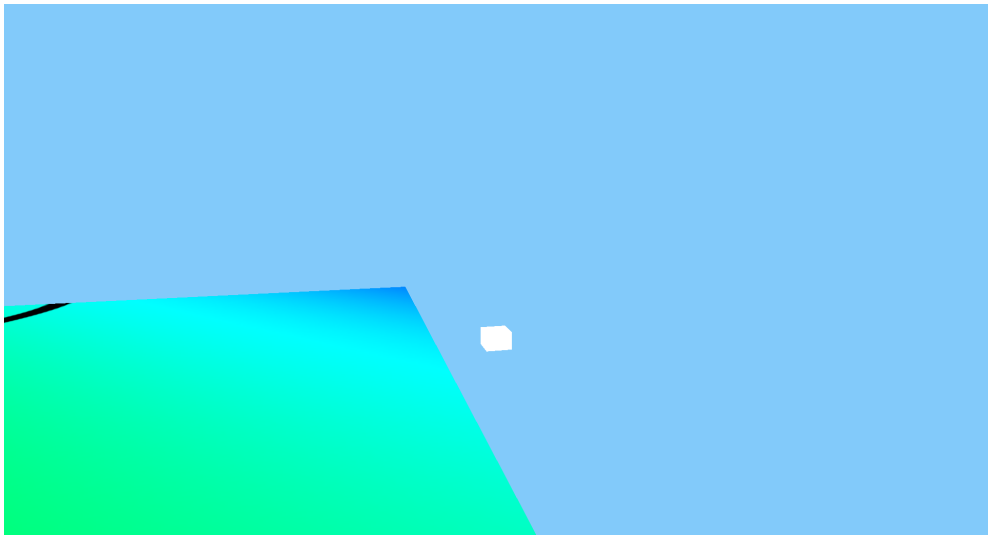
```
void scale_model(Model *model, float scalex, float scaley,
float scalez)
{
    int k = 0;
    while (k < model->i_f)
    {
        for (int i = 0; i < 3; i++)
        {
            model->v[k].x *= scalex;
            model->v[k].y *= scaley;
            model->v[k].z *= scalez;
        }

        k++;
    }
}
```

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 4.2, illetve 4.3 ábrákon láthatunk.



4.2. ábra. Alapértelmezett playermodell méret

⇓

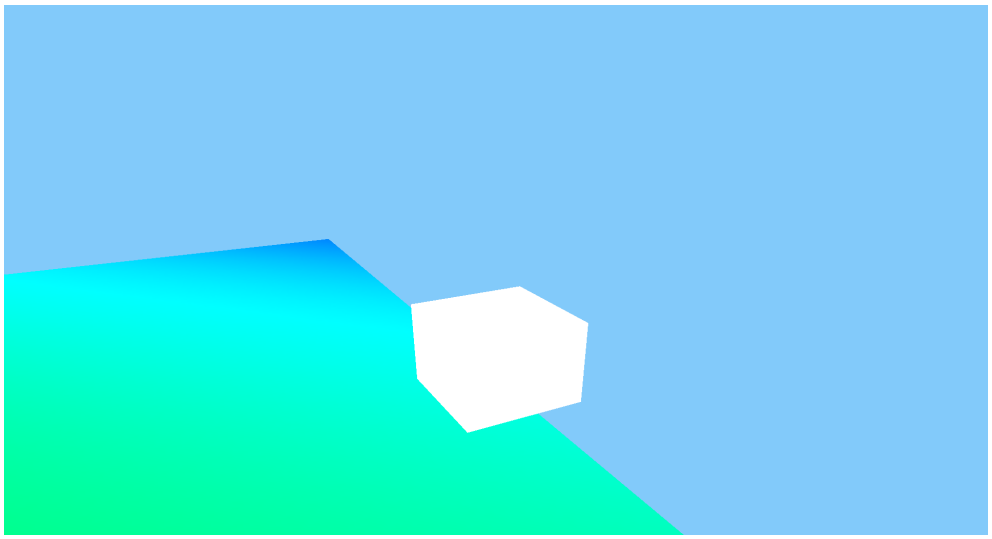


4.3. ábra. Playermodell méretének csökkentése

4.3.2. Modell forgatása

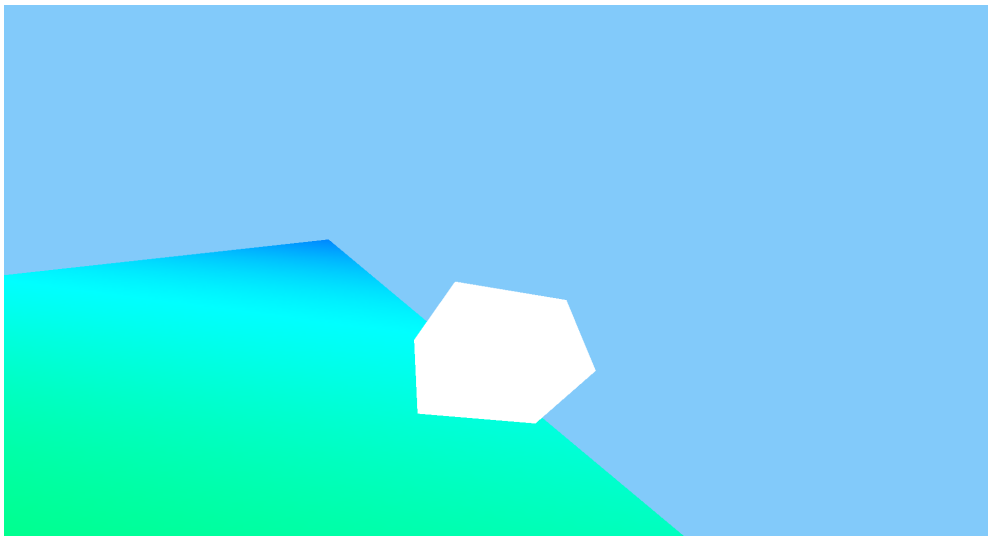
```
void rotate_model(Model *model, float anglex, float angley, float anglez)
{
    int k = 0;
    float cord1, cord2, angle;
    while (k < model->i_f)
    {
        cord1 = model->v[k].x;
        cord2 = model->v[k].y;
        angle = anglex * (float)(M_PI / 180);
        model->v[k].x = cord1 * cosf(angle)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

A modell forgatása az egyik legnehezebb feladat az összes közül. Itt nem elég szimplán szoroznunk, hanem a forgatás miatt **cos**inus, illetve **sin**us számításokat is végeznünk kell. A függvény végigmegy minden egyes háromszögén a modellnek, majd a csúcspontok koordinátáit módosítja **sin**us és **cos**inus számítással az adott forgatási szögek alapján. Modellek forgatásáról példát a 4.4, illetve 4.5 ábrákon láthatunk.



4.4. ábra. Alapértelmezett playermodell forgatás

⇓



4.5. ábra. Playermodell forgatása a tengelyeken

5. fejezet

Optimalizálás

Minél több a "playermodell" háromszögeinek száma, illetve a többi modellek száma, és azok háromszögeinek száma, annál több számítást kell végeznünk. Például 2 modell esetén, ha a playermodellünk 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. Emiatt a módszer nem alkalmas valós idejű alkalmazásokhoz, viszont cserébe annál alkalmasabb olyan alkalmazásokhoz, ahol fontos a nagy pontosságú ütközésvizsgálat alkalmazása. Több módszer is ki lett próbálva lehetséges optimalizálás szempontjából.

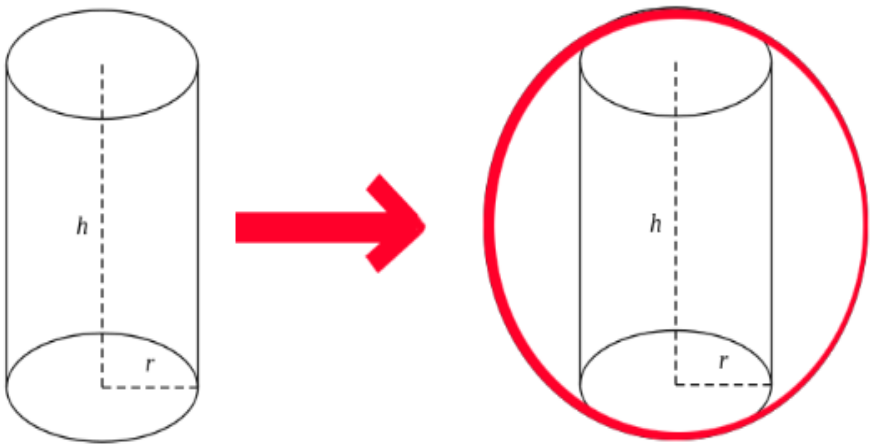
5.1. Távolság alapján szűrés

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.
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;
}
```

A *limit* változóban megnézzük a maximális távolságot, amelyen belül már ütközésvizsgálatot vizsgálunk. Összeadjuk a két modell legtávolabbi pontjának távolságát, illetve egy extra konstant, hogy biztosra menjünk. 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.



5.1. ábra. Modell köré egy gömb alakú "hitbox" generálása

5.2. Távolság szűrése háromszögekkel

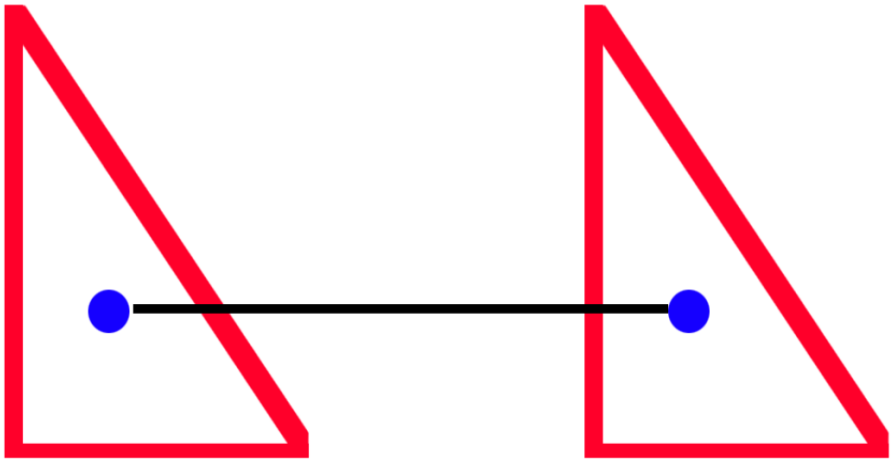
Lényege, hogy ütközésvizsgálat előtt a modellek háromszögeinek a középpontját kiszámítjuk, majd a középpontokat viszonyítjuk egymáshoz, ahogy a 5.2 ábrán láthatjuk. Ha a háromszögek közötti távolság nagyobb, mint a maximum táv, akkor az adott háromszög számítása elhanyagolható.

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 alaphoz képest, de csökken az előző algoritmushoz képest, illetve megnő a memóriaigény is. Használata nem ajánlott.

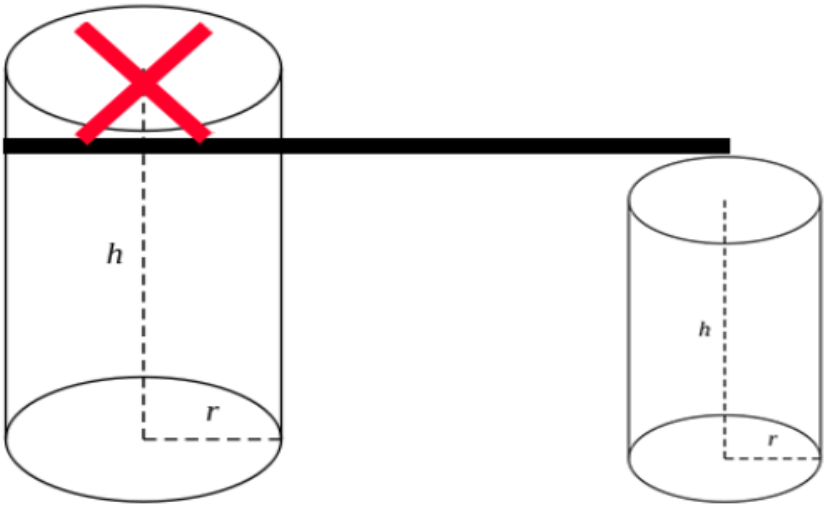


5.2. ábra. Távolság szűrése háromszögek középpontjával

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 a 5.3 ábrán láthatjuk. Például ha a playermodellünk a Z koordinátán 20 cm magas, akkor a maximum szint a Z tengelyen az a modell középpontja + 20 cm 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, viszont programozás terén nehezen kivitelezhető, sok esetben hibákhoz vezethet, a memória igény nő.



5.3. ábra. Tengelyek alapján szűrés

5.5. OpenCL

A C nyelv OpenCL függvénykönyvtárának használatával a bonyolultabb számításokat a számítógépünk videókártyájával számíthatjuk ki a processzor helyett. Így sokkal gyorsabban tudunk 1-1 nagyobb számítást elvégezni.

Az optimalizálás ezen esetben **sikertelen**, a számítások ideje csökken, de az átlag FPS a programon belül csökken, mivel túl sok ideig tart 1-1 kernel hívás a számítások elvégzéséhez.

6. fejezet

Összefoglalás

A program könnyen használható olyan programozók számára is, akik nem teljesen jártasak az ütközésvizsgálat terén, és egy könnyen kezelhető függvénykönyvtárat szeretnének használni.

Csak importálni kell a függvénykönyvtárat, és máris működőképesek a beépített funkciók. A programozó maga tudja eldönteni, hogy mikor, vagy éppen melyik modellekre érvényesek az ütközések vizsgálata.

Vizsgálatonként elég meghívni a `check_collision` funkciót, átadni a szükséges adatokat, és máris visszakapjuk az adott értéket, hogy 2 modell ütközik-e egymással vagy sem.

A jövőben további fejlesztési lehetőségek egyike lehet a program optimalizálása, vagy további egyszerűsítése.

Irodalomjegyzék

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

CD Használati útmutató

A függvénykönyvtár használatához szükségünk van egy OpenGL 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.