



Budapesti Műszaki és Gazdaságtudományi Egyetem
Villamosmérnöki és Informatikai Kar
Irányítástechnika és Informatika Tanszék

Fizikai motor fejlesztése Rust nyelven

SZAKDOLGOZAT

Készítette
Szarkowicz Dániel

Konzulens
Fridvalszky András

2024-12-03

Tartalomjegyzék

Kivonat	1
1. Bevezetés	2
1.1. Merev test	2
1.1.1. Merev testek tulajdonságai	2
1.1.2. Merev test szimuláció	2
2. Fizikai motorok	3
2.1. Box2D	3
2.2. Rapier	3
3. Rust	4
4. Merevtestek matematikája	5
4.1. Lineáris mozgás	5
4.2. Forgás	5
4.3. Ütközésválasz	6
4.3.1. Impulzus és szögimpulzus	6
4.3.2. Lokális sebesség és lokális tehetetlenség	7
4.3.3. Normál irányú impulzus	7
4.3.4. Nem normál irányú (súrlódási) impulzus	7
5. Ütközés detektálás	9
5.1. Egyszerű gömb ütközés	9
5.2. GJK	10
5.3. EPA	11
5.4. Ütközési pontok kiszámítása	12
5.4.1. Oldalak kiszámítása	12
5.4.2. Oldalak metszete	13
6. Ütközés detektálás gyorsítása	14
6.1. Sort-and-Sweep	14
6.2. R-Tree	15
6.3. OMT	16
6.3.1. Az algoritmus	16

7. Nyugalmi érintkezés	18
7.1. Analitikus megoldás	18
7.2. Iteratív megoldás	19
8. Fizikai könyvtár	20
8.1. Példa kód	20
9. Eredmények	22
9.1. Gyorsítási módszerek összehasonlítása	22
Függelék	23
A. OMT vágások kiszámítása	23
Irodalomjegyzék	25

HALLGATÓI NYILATKOZAT

Alulírott *Szarkowicz Dániel*, szigorló hallgató kijelentem, hogy ezt a szakdolgozatot meg nem engedett segítség nélkül, saját magam készítettem, csak a megadott forrásokat (szakirodalom, eszközök stb.) használtam fel. Minden olyan részt, melyet szó szerint, vagy azonos értelemben, de átfogalmazva más forrásból átvettem, egyértelműen, a forrás megadásával megjelöltem.

Hozzájárulok, hogy a jelen munkám alapadatait (szerző(k), cím, angol és magyar nyelvű tartalmi kivonat, készítés éve, konzulens(ek) neve) a BME VIK nyilvánosan hozzáférhető elektronikus formában, a munka teljes szövegét pedig az egyetem belső hálózatán keresztül (vagy autentikált felhasználók számára) közzétegye. Kijelentem, hogy a benyújtott munka és annak elektronikus verziója megegyezik. Dékáni engedéllyel titkosított diplomatervek esetén a dolgozat szövege csak 3 év eltelte után válik hozzáférhetővé.

Budapest, 2024-12-03

Szarkowicz Dániel
hallgató

Kivonat

A realisztikus, valós idejű szimulációknak egyre fontosabb szerepe van a számítógépes grafika terén. Az szimulációk új lehetőségeket nyitnak a videójátékok, animációk és tervezőprogramok területén.

Ebben a dolgozatban megismerkedünk néhány elterjedt fizikai motorral, a merevtestek mozgásának matematikájával, a Gilbert-Johnson-Keerthi algoritmussal és bővítéseivel, majd megvizsgálunk néhány gyorsítási módszert, amiknek köszönhetően akár több ezer testet szimulálhatunk.

A dolgozat végén implementálunk egy fizikai motort a Rust programozási nyelven, amelyet néhány egyszerű példával mutatunk be.

1. Fejezet

Bevezetés

1.1. Merev test

A newtoni fizikában a merev test a szilárd testnek egy ideális modellje. A merev testeknek nem változhat meg az alakja sem külső, sem belső hatásoktól.

1.1.1. Merev testek tulajdonságai

Tömeg a mozgásállapot változással szembeni ellenállás mértéke

Sebesség egy pillanatban a merev test mozgásállapot változásának mértéke

Lendület a merev test sebessége összeszorozva a tömegével

Lendület megmaradás egy merev test lendülete csak külső hatásra változhat meg

Impulzus a lendület megváltozása külső hatásra

Tehetetlenségi nyomaték a forgás változással szembeni ellenállás mértéke

Szögsebesség egy pillanatban a merev test forgásának változásának mértéke

Perdület a tehetetlenségi nyomaték és a szögsebesség szorzata

Perdület megmaradás egy merev test perdülete csak külső hatásra változhat meg

Szögimpulzus a perdület megváltozása külső hatásra

1.1.2. Merev test szimuláció

Az informatikában a merevtest-szimuláció egy elterjedt módszer a hétköznapi testek szimulálására. Önvezető autók tanításánál az járművek szimulálására, a videójátékok terén az imerzivitás növelésére használhatják az élethű, valós idejű szimulációkat.

2. Fejezet

Fizikai motorok

2.1. Box2D

A Box2D egy 2 dimenziós fizikai motor, amelyet a C programozási nyelvben írtak. A motor sok féle szimulációt támogat:

- ütközés
- tapadási és csúszási súrlódás
- jointok
- ragdollok
- ray casting

A motor egy iteratív megoldást használ a korlátok megoldására, és így is jó minőségű stabil szimulációkat tud produkálni. Sok objektumot képes valós időben szimulálni, így számos videojáték használja.

Néhány híres videojáték:

- Angry Birds
- Happy Wheels
- Limbo
- Showel Knight

A Box2D-t egy egyszerű C API-n keresztül lehet használni, számos nyelvhez készültek hozzá binding-ok és a legtöbb játékmotorban is használható beépítve vagy egy bővítményként.

2.2. Rapier

A Rapier egy 2 és 3 dimenziós fizikai motor, amelyet a Rust programozási nyelven írtak. A motor hasonló funkciókat támogat, mint a Box2D. Nagy hangsúlyt helyez a determinizmusra, akár különböző architektúrájú platformokon is.

A Rapier-t egy egyszerű Rust API-n keresztül lehet használni és számos modern játékmotorban használható.

3. Fejezet

Rust

A Rust egy modern, nagy teljesítményű rendszerprogramozási nyelv, amely nagy hangsúlyt helyez a program memóriabiztonságára és helyességére. A memóriabiztonság garantálására fordítás (pl. RAII) és futtatás idejű (pl. tömb méret indexeléskor) ellenőrzéseket használ. A program helyességét egy erős típus rendszer és a borrow checker garantálja. Néha teljesítményi okokból olyan kódot írunk, amirő a fordító nem tudja belátni, hogy helyes, ilyenkor az `unsafe` kulcsszóval tudunk olyan metódusokat (pl. `get_unchecked`) és típusokat (pl. `UnsafeCell`) használni, amelyek nem garantálják a program helyességét.

A Rusthoz egy nagyon nagy ökoszisztéma is tartozik amelyekkel könnyen lehet cross-platform alkalmazásokat fejleszteni. A kirajzoláshoz felhasznált `wgpu` könyvtár például egy alacsony szintű absztrakciós réteget biztosít a különböző grafikus API-k között, így a programot viszonylag egyszerű akár weben is futtatni.

4. Fejezet

Merevtestek matematikája

4.1. Lineáris mozgás

Egy test mozgásának a leírásához a test pozíciójára és sebességére lesz szükség. A pozícióra és a sebességre a következőt írhatjuk fel:

$$v(t) = \frac{d}{dt}x(t)$$

A szimuláció állapota időben diszkrét módon frissül. Az új állapotot a következő módon számolhatjuk ki az előző állapotból:

$$x(t + \Delta t) = x(t) + \Delta t \cdot v(t)$$

Ezt a módszert Euler integrációnak hívjuk. Léteznek pontosabb számítási módszerek is, például a Runge-Kutta metódus.

A szimuláció a test sebessége helyett a test lendületét tárolja, ez a következő módon áll kapcsolatban a sebességgel:

$$p(t) = m \cdot v(t)$$

Ennek az előnyeiről bővebben az 4.3 fejezetben fogok írni.

4.2. Forgás

A testek forgása a test mozgásához hasonlóan kezelhető. A testnek van egy elfordulása, amit egy forgatásmátrixban tárolunk és egy szögsebessége, amit egy tengellyel és egy nagysággal jellemzünk, ez egy vektorban tárolható.

A testnek az új elfordulása a következő módon számolható ki a régi elfordulásból és a szögsebességéből:

$$R(t + \Delta t) = (\Delta t \cdot \omega(t)^*) \cdot R(t),$$

$$\text{ahol [1] szerint } \omega(t)^* = \begin{pmatrix} 0 & -\omega(t)_z & \omega(t)_y \\ \omega(t)_z & 0 & -\omega(t)_x \\ -\omega(t)_y & \omega(t)_x & 0 \end{pmatrix}$$

Míg a mozgásnál a lendületmegmaradás általában megegyezik a sebességmegmaradással, a forgásnál a perdületmegmaradás nem egyezik meg a szögsebesség-megmaradással, mert a tehetetlenségi nyomaték nem konstans. A newtoni mechanika szerint perdületmegmaradás van, ezért a szimulációban érdemes a szögsebesség helyett a perdületet tárolni. A perdület a következőképpen áll kapcsolatban a szögsebességgel:

$$L(t) = \Theta(t) \cdot \omega(t),$$

$$\text{ahol [1] szerint } \Theta(t) = R(t) \cdot \Theta \cdot R(t)^{-1}$$

Egy testnek az alap tehetetlenségi nyomatéka az alakjától és a súlyeloszlásától függ. A szimulációban használt testek tehetetlenségi nyomatéka a következő:

$$\Theta_{\text{gömb}} = \frac{2}{3}m \cdot r^2$$

$$\Theta_{\text{téglatest}} = \frac{m}{12} \cdot \begin{pmatrix} h^2 + d^2 & 0 & 0 \\ 0 & d^2 + w^2 & 0 \\ 0 & 0 & w^2 + h^2 \end{pmatrix}$$

A szimulációban a tehetetlenségi nyomatéknak csak az inverzét használjuk, mert mindig perdületből konvertálunk szögsebességbe, ezért a tehetetlenségi nyomatéknak az inverzét tárolja.

4.3. Ütközésválasz

4.3.1. Impulzus és szögimpulzus

A szimulációban a testek nem deformálódhatnak és nem metszhetik egymást, ezért az ütközésnek egy pillanatnyi eseménynek kell lennie. Mivel az erő és a forgatónyomaték 0 idő alatt nem tudnak változást elérni, ezért helyettük impulzusokat és szögimpulzusokat kell használni.

A impulzus kifejezhető úgy, mint egy erő, ami egy kicsi idő alatt hat:

$$J = F \cdot \Delta t$$

Ha az impulzus egy x_J pontban hat a testre, akkor a szögimpulzus:

$$M = (x_J - x(t)) \times F, \quad \text{a forgatónyomaték}$$

$$\Delta L = M \cdot \Delta t = (x_J - x(t)) \times F \cdot \Delta t = (x_J - x(t)) \times J$$

Tehát, ha egy testre egy J impulzus hat egy x_J pontban, akkor a test lendülete és perdülete a következő módon változik meg:

$$p'(t) = p(t) + J$$

$$L'(t) = L(t) + (x_J - x(t)) \times J$$

4.3.2. Lokális sebesség és lokális tehetetlenség

Az ütközési számításokhoz szükséges lesz a testek lokális sebességére egy adott x_J pontban. Ez a sebességből és a szögsebességből származó kerületi sebesség összege:

$$v_l = v(t) + \omega(t) \times (x_J - x(t)) = \frac{p(t)}{m} + (\Theta^{-1}(t) \cdot L(t)) \times (x_J - x(t))$$

Szükség lesz még a testek lokális tehetetlenségére. Ez a test tömegéből és tehetetlenségi nyomatékából származó ellenállás a lokális sebesség adott irányú változására. Ennek az inverzét így számoljuk ki [2] egy adott irányban az x_J pontban:

$$T^{-1}(\hat{u}) = \hat{u} \cdot \left[\frac{\hat{u}}{m} + \left(\Theta^{-1}(t) \cdot [(x_J - x(t)) \times \hat{u}] \right) \times (x_J - x(t)) \right]$$

4.3.3. Normál irányú impulzus

Két test ütközésekor a testek lokális relatív sebességével (v_{lr}) kell számolni.

$$v_{lr} = v_{1,l} - v_{2,l}$$

Jelölje az ütközés utáni lokális relatív sebességet v'_{lr} .

Ha két test tökéletesen rugalmasan ütközik, akkor v'_{lr} normál irányú komponense v_{lr} normál irányú komponensének a negáltja lesz:

$$v'_{lr} = v_{lr} - 2\hat{n} \cdot (\hat{n} \cdot v_{lr})$$

Ha két test tökéletesen rugalmatlanul ütközik, akkor v'_{lr} normál irányú komponense 0 lesz:

$$v'_{lr} = v_{lr} - \hat{n} \cdot (\hat{n} \cdot v_{lr})$$

Jelölje ε az ütközés rugalmasságát. Ha $\varepsilon = 1$, akkor az ütközés tökéletesen rugalmas, ha 0 akkor tökéletesen rugalmatlan. Így a normál irányú sebességnek a változása a következő:

$$\Delta v_{lr,n} = -(1 + \varepsilon) \cdot (\hat{n} \cdot v_{lr})$$

Ezt a sebességváltozást a lendületmegmaradás törvénye értelmében egy azonos nagyságú, ellentétes irányú impulzus fogja kiváltani a két testen. Az impulzus nagysága a testek normál irányú lokális tehetetlenségéből jön ki [2]:

$$|J_n| = \frac{\Delta v_{lr,n}}{T_1^{-1}(\hat{n}) + T_2^{-1}(\hat{n})}$$

4.3.4. Nem normál irányú (súrlódási) impulzus

Két test ütközése során nem csak normál irányú erők (impulzusok) hatnak a testekre, mert a testek súrlódnak is egymáson. A súrlódás „célja” az, hogy a testek v_{lr} -ének a nem normál irányú komponenseit 0 felé közelítse.

A súrlódási impulzus nagyságának maximuma a normál irányú impulzus nagyságától és a súrlódási együtthatótól függ:

$$|J_s| \leq |J_n| \cdot \mu$$

A súrlódási impulzus irányához és nagyságához kelleni fog a v_{lr} nem normál irányú komponense:

$$v_{lr,s} = v_{lr} - \hat{n} \cdot (\hat{n} \cdot v_{lr})$$

A sebesség 0-ra állításához szükséges impulzus nagyságát a következő módon kaphatjuk meg:

$$|J_s^+| = \frac{-|v_{lr,s}|}{T_1^{-1}(\hat{v}_{lr,s}) + T_2^{-1}(\hat{v}_{lr,s})}$$

Tehát a súrlódási impulzus:

$$J_s = \min(|J_s^+|, |J_n| \cdot \mu) \cdot \hat{v}_{lr,s}$$

A súrlódást külön lehet bontani tapadási és csúszási súrlódásra. Ilyenkor ha $|J_s^*|$ nagyobb, mint a tapadási súrlódás maximális nagysága, akkor a csúszási súrlódás nagyságát használjuk a súrlódási impulzus nagyságaként.

5. Fejezet

Ütközés detektálás

5.1. Egyszerű gömb ütközés

A szimuláció eleinte csak gömböket támogatott, mert azokra a legegyszerűbb kiszámolni, hogy ütköznek-e.

Két gömb akkor ütközik, ha a középpontjaik távolsága kisebb, mint a sugaraik összege:

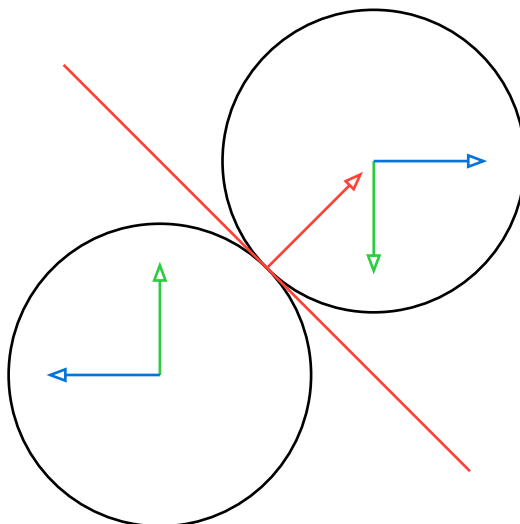
$$|c_1 - c_2| < r_1 + r_2$$

Ha ütköznek, akkor az ütközési normál:

$$n = \frac{c_1 - c_2}{|c_1 - c_2|}$$

és az ütközési pontok:

$$p_1 = c_1 - r_1 n$$
$$p_2 = c_2 + r_2 n$$

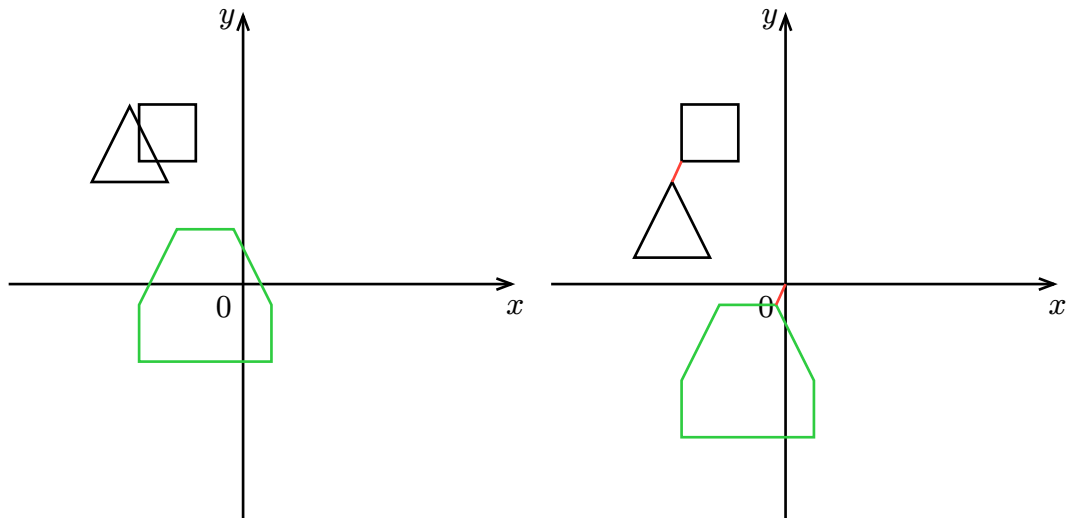


Ábra 1: Két gömb ütközik, **piros** az ütközési felület és normál, **zöld** az ütközés előtti sebességek, **kék** az ütközés utáni sebességek

5.2. GJK

Egy elterjedt ütközés detektálási algoritmus a Gilbert-Johnson-Keerthi [3] algoritmus, ami tetszőleges konvex testek távolságát tudja meghatározni, ha a testekre definiálva van a support function. A support functionnek egy adott irányban kell a test legtávolabbi pontját visszaadni.

Az algoritmus két test Minkowski különbségéről vizsgálja meg, hogy benne van-e az origó. Ha a különbségben benne van az origó, akkor ütközést talált, ha a különbségben nincs benne az origó, akkor Minkowski különbség és az origó távolsága a két test távolsága. A Minkowski különbség legközelebbi pontjából ki lehet fejteni a két test legközelebbi pontját is.



Ábra 2: Egy háromszög és egy négyzet Minkowski különbsége. Az első képen a két test ütközik, a Minkowski különbségük tartalmazza az origót. A második képen a két test nem ütközik, a távolságuk megegyezik a Minkowski különbség és az origó távolságával.

A Minkowski különbség összes pontján egy kicsit sok időbe telne végig iterálni, de a különbséget felépítő simplexein (2 dimenzióban háromszög, 3 dimenzióban tetraéder) már lehetséges. Ehhez a Minkowski különbség support functionjére lesz szükség, amit a következő módon számíthatunk ki egy A és egy B test support functionjéből:

$$s(\mathbf{d}) = s_A(\mathbf{d}) - s_B(-\mathbf{d})$$

Tehát az algoritmus a Minkowski különbség simplexein iterál végig. Ezekkel a simplexeikkel egyre közelíteni szeretnénk az origót, míg vagy a simplex tartalmazza az origót, vagy nem sikerült közelebb jutnunk az origóhoz. Az origóhoz úgy lehet közelíteni, hogy a simplexnek vesszük az origóhoz a legközelebbi részsimplexét és a legközelebbi pontját, és a legközelebbi ponttal ellentétes irányba kérünk a support functiontól egy új pontot, amit hozzáadunk a simplexhez.

Az algoritmus kétféleképpen került implementációra.

Az első implementáció baricentrikus koordinátákkal kereste meg a legközelebbi részsziimplexet és a legközelebbi pontot. Sajnos a legközelebbi részsziimplex keresésénél néha nem egyértelmű, hogy melyik részsziimplex van közelebb és többet is meg kell vizsgálni, ami nem csak azért probléma, mert több számítást végez, de azért is, mert így másolni kell a szimplex adatait, amit nem lehet kioptimalizálni.

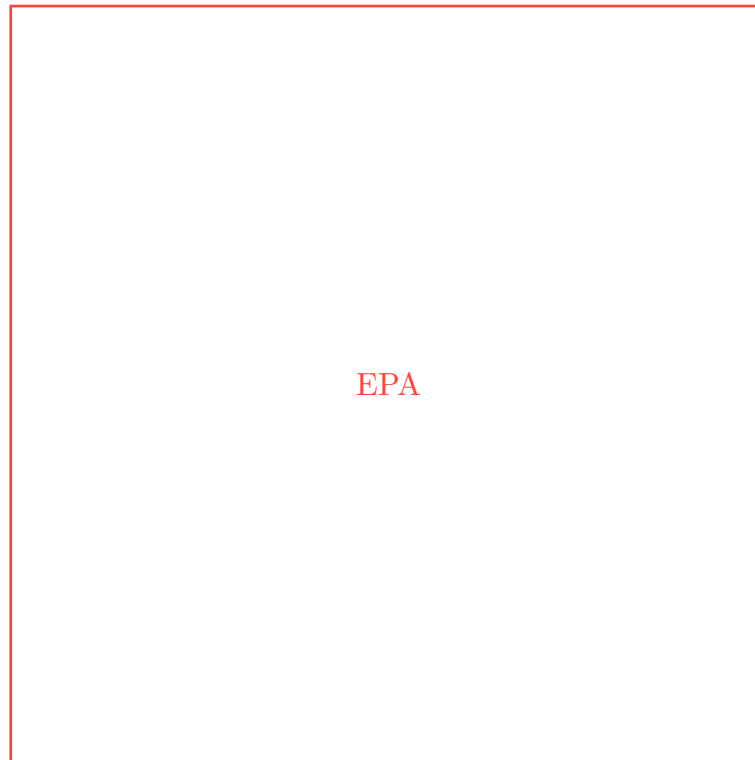
A második implementáció a teret részsziimplexenként két részre osztja és megnézi, hogy a részsziimplexen belül vagy kívül esik-e az origó. Előbb vagy utóbb egy néhány részsziimplex vagy körbe fogja az origót, és akkor tudjuk, hogy a részsziimplexek által alkotott szimplex tartalmazza az origóhoz a legközelebbi pontot, vagy egy ponton (0 dimenziós szimplex) kívül esik az origó, és akkor tudjuk, hogy a pont a legközelebbi pont (és részsziimplex) az origóhoz. A szimuláció [4] által bemutatott 2 dimenziós algoritmusnak egy 3 dimenziós generalizációját használja.

A GJK könnyen használható gömbileg kiterjesztett testekre, például egy gömbre vagy kapszulára, hiszen a két test legközelebbi pontja adott és sugara adott, inentől a 5.1 fejezetben írt módon lehet kiszámolni, hogy a két test ütközik-e, és ha igen, akkor mik az ütközési paraméterei.

5.3. EPA

A GJK egyik hiányossága, hogy ha két test ütközik, akkor csak annyit mond, hogy ütköznek, nem ad nekünk használható ütközési paramétereket. Az EPA úgy segít, hogy a GJK-ból kapott szimplexet iteratívan bővíti újabb pontokkal, amíg megtalálja az átfedő terület szélességét. Az EPA a GJK-ban használt legközelebbi pont algoritmust használja, de nem az egész politópon futtatja, hanem csak a politóp oldalait alkotó szimplexeken.

Az EPA a Minkowski különbségnek az origóhoz legközelebbi felszíni pontját keresi meg. Ezt úgy éri el, hogy a politóp legközelebbi pontjának irányában kér egy új pontot a különbség support functionjétől, ha talált távolabbi pontot, akkor kiegészíti a politópot az új ponttal, ha nem talált távolabbi pontot, akkor megtaláltuk a Minkowski különbség legközelebbi felszíni pontját.



Ábra 3: Az EPA felfedte a Minkowski különbség egy részét, amíg megtalálta a legközelebbi felszíni pontot.

A politóp bővítése nem egy könnyű feladat, ugyanis ha hozzáadunk egy új pontot a politóphoz, akkor ki kell számolni, hogy milyen régi oldalakat kell kitörölni, és hogy milyen új oldalakat kell felvenni. Azokat a régi oldalakat kell törölni, amelyek az új pont „alatt” vannak, azaz az egyik oldalukon az új pont van, a másik oldalukon pedig az origó. Az új oldalakat úgy kell hozzáadni, hogy a régi oldalak azon széleit, amelyeket csak az egyik oldalról határolt kitörölt oldal összekötjük az új ponttal. Ez a bővítés elképzelhető egy konvex burok iteratív felépítéseként is.

A szimuláció [5] által bemutatott 2 dimenziós algoritmusnak egy 3 dimenziós generalizációját használja.

5.4. Ütközési pontok kiszámítása

A GJK és az EPA csak egy ütközési pontot adnak, amely pillanatnyi érintkezésnél elfogadható, de **Nyugalmi érintkezés**-nél nem.

Több ütközési pontot úgy kaphatunk, hogy az ütközési normál mentén lekérjük a ütköző testeknek „legjobb” oldalait, ezeknek az oldalakna vesszük a meteszetét az ütközési normál szerint és a metszet pontjaiból választunk néhányat az ütközési pontoknak.

5.4.1. Oldalak kiszámítása

A „legjobb” oldalak kiszámítása a test alakjától függ.

Egy gömbnek a legjobb oldala a gömb középpontjából és sugarából könnyen kiszámolható.

Egy kocka legjobb oldalához ki kell számolni minden oldal normálvektorának a szögét az ütközési normállal, kiválasztjuk a legkisebb szöget és a hozzá tartozó oldalt adjuk vissza.

5.4.2. Oldalak metszete

Az oldalak metszetéhez egy közös síkra kell vetíteni az oldalakat, utána egy 2 dimenziós algoritmussal vesszük az oldalak metszetét, végül a normál vektor mentén visszavetítjük az oldalakat a saját síkjukra.

Normál vektor síkjára vetítés:

$$p_n = p + (p_0 - p) \cdot \hat{n} \cdot \hat{n}$$

Eredeti síkra vetítés a normál vektor mentén:

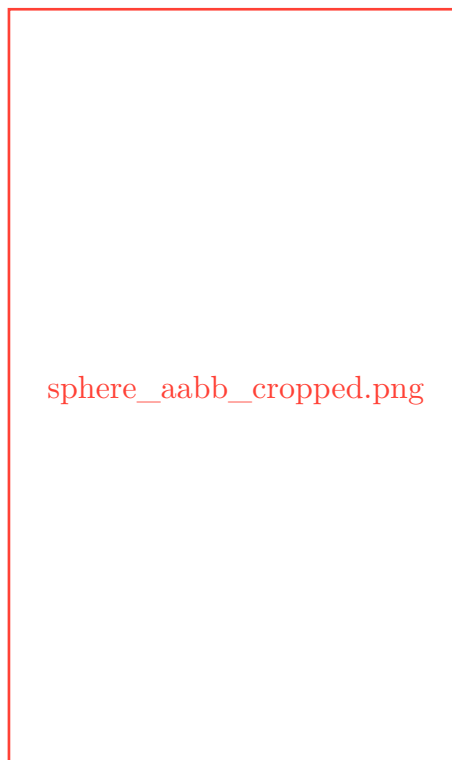
$$p' = p'_n - \frac{(p'_n - p'_0) \cdot \hat{n}'}{\hat{n} \cdot \hat{n}'} \cdot \hat{n}$$

Az oldalak metszetének kiszámolásához a **clipper2**-t használtam.

6. Fejezet

Ütközés detektálás gyorsítása

Az összes pár megvizsgálása $O(n^2)$ lenne, ami nagyon lassú. Szerencsére a legtöbb test nem ütközik, ezért egy megfelelő heurisztikával sokat lehet spórolni. A szimuláció gyorsításához kell egy algoritmus, ami gyorsan eldobja a teszteknek egy jelentős részét és így csak a párok egy kis hányadát kell megvizsgálni. Ezek az algoritmusok általában csak egyszerű alakzatokon tudnak dolgozni, a következő algoritmusok axis-aligned bounding boxokat (AABB) használnak. Az AABB-k olyan téglatestek, amik tartalmazzák az az egész testet és az oldalai párhuzamosak a koordináta-rendszer tengelyeivel.

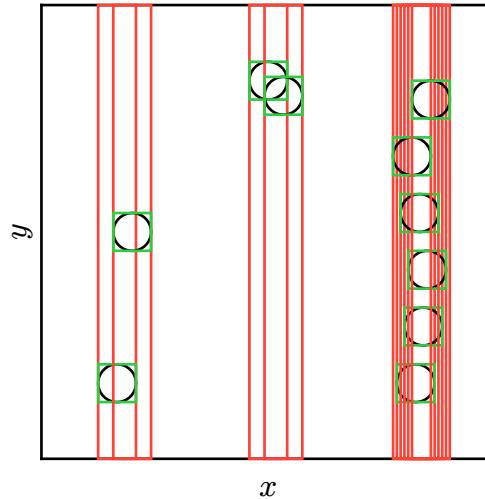


Ábra 4: Egy gömbnek az AABB-je.

6.1. Sort-and-Sweep

A sort and sweep [2] egy egyszerű algoritmus az ellenőrzött párok csökkentésére. Az algoritmus az egyik tengely szerint intervallumokként kezeli az AABB-ket és átfedő

intervallumokat keres. Ezt úgy éri el, hogy egy listába kigyűjti minden AABB-nek az elejét és a végét a kiválasztott tengely szerint, rendezi a listát, majd egyszer végig iterál a listán és kigyűjti az átfedő intervallumokat. Az átfedő intervallumok listája tartalmazza a potenciális ütközéseket, ezt a listát érdemes szűrni az AABB-k másik két tengelye szerint, mielőtt a tényleges ütközés detektálás algoritmust futtatnánk. Az algoritmusnál sokat számíthat a megfelelő tengely kiválasztása, rossz tengely megválasztásakor lehet, hogy csak a pároknak egy kis részét dobjuk el.



Ábra 5: A sort and sweep algoritmus intervallumai az x tengely szering. A jobb oldalon látható, hogy ha rossz tengelyt választunk, akkor nem segít sokat az algoritmus.

6.2. R-Tree

A szimulációban használt broad phase algoritmus (adatstruktúra) az R-Tree. Az R-Tree a B-Tree-nek egy kibővített változata, ami több dimenzió szerint rendezhető adatokra tud hatékony keresést biztosítani.

A R-Tree minimal bounding box-okból épül fel. Ezek olyan AABB-k, amiknél kisebb AABB nem lenne képes bennfoglalni a tartalmazott elemeket. Az R-Tree-be felépítéskor egyesével illesztjük be az AABB-ket. Az R-Tree-nél két fontos algoritmus van: legjobb csúcs kiválasztása a beillesztéshez, és legjobb vágás kiszámítása, ha egy csúcs megtelt. A szimuláció a beillesztéshez a legkisebb térfogat növekedést választja, a vágáshoz a Quadratic split-et [6] használ.



Ábra 6: Egy 3 dimenziós R-Tree kettő szintje. Látható, hogy néha átfedik egymást a csomópontok (piros AABB-k). Az átfedéseket érdemes minimalizálni.

Az R-Tree-k felépítéséhez és karbantartásához számos algoritmus jött létre.

Az R*-Tree [6] optimálisabb csúcsválasztást és optimálisabb vágást használ és ha a vágás után egy elem nagyon nem illik be egyik csoportba sem, akkor újra beilleszti a fába, hátha talál jobb helyet.

Az STR [7] és az OMT [8] nem egyesével építi fel a fát, hanem egyszerre dolgozik az összes adattal, így közel tökéletes fákat tudnak felépíteni.

6.3. OMT

Az Overlap Minimizing R-Tree egy bulk-loading algoritmus, amely az eredeti R-Tree felépítésével ellentétben nem egyesével építi fel a fát, hanem egyszerre.

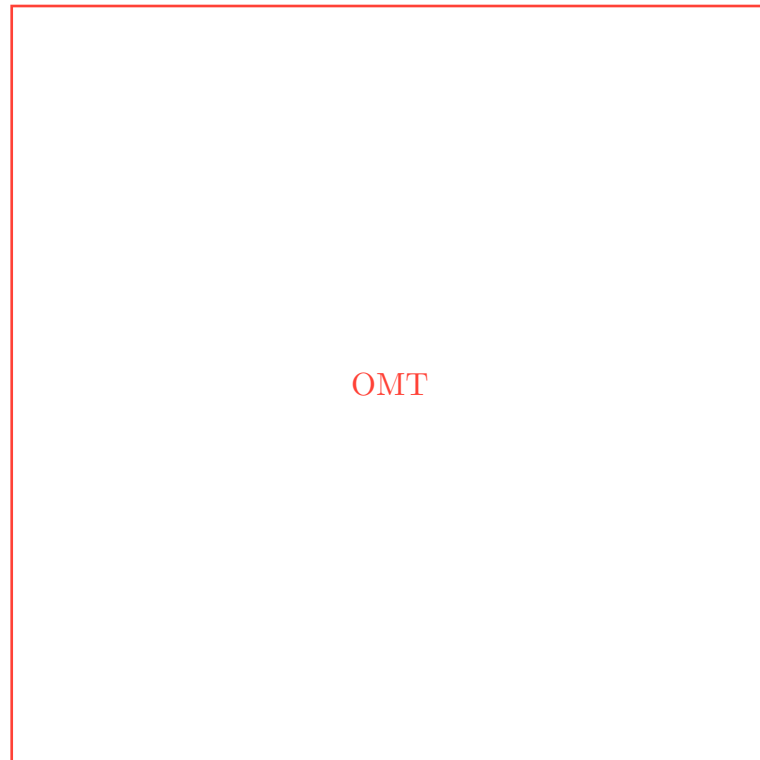
Ezzel a módszerrel hatékonyabb keresőfát tudunk felépíteni, ugyanis az algoritmus törekszik minimalizálni a részek között az átfedést.

6.3.1. Az algoritmus

Az algoritmus egy lépés ismételt addig, amíg a legalsó csúcsához tartozó levelek száma kisebb, mint egy határ érték. Ez a lépés a csúcs leveleit a következő módon bontja N egyenlő részre:

1. vágások kiszámítása mindhárom dimenzió szerint (A. függelék)
2. rendezés az első dimenzió szerint
3. vágás az első dimenzió szerint
4. vágások rendezése a második dimenzió szerint
5. vágás a második dimenzió szerint

6. vágások rendezése a harmadik dimenzió szerint
7. vágás a harmadik dimenzió szerint



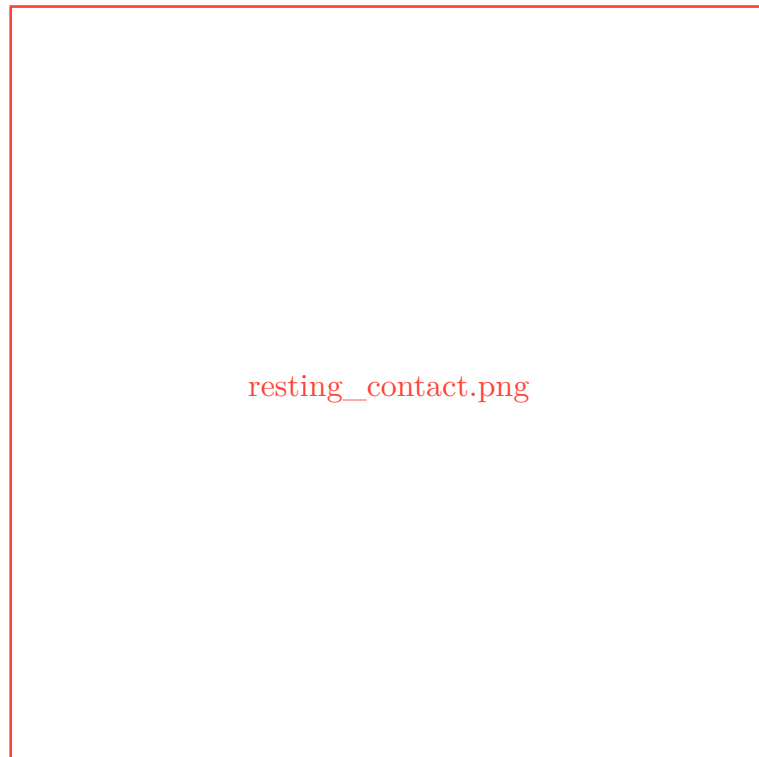
Ábra 7: Az Overlap Minimizing R-Tree. Látható, hogy sokkal kevesebb az átfedés, mint az R-Tree-ben.

Az OMT-vel generált fát az összes többi R-Tree algoritmussal használhatjuk tovább. A fizikai motor csak a keresést használja az R-Tree-ből, mert jelenleg minden ciklusban újraépíti a fát.

7. Fejezet

Nyugalmi érintkezés

Ha a testekre erők is hatnak (nem csak impulzusok), akkor könnyen előfordulhat, hogy a testek hosszabb ideig érintkeznek. Ezeknél az érintkezéseknél külön oda kell figyelni, hogy a testek ne csússzanak egymásba, de itt nem használhatunk nagy erőket a testek szétválasztásához, mert akkor a szimuláció nem lenne élethű.



Ábra 8: Nyugalmi érintkezéskor a testek relatív sebessége 0, oda kell figyelni, hogy a relatív sebesség ne csökkenjen.

Ebben a dolgozatban két megoldást fogunk megvizsgálni.

7.1. Analitikus megoldás

Az analitikus megoldásban egyszerre számoljuk ki az összes érintkezési pontban az erőket, így tökéletes megoldást kaphatunk. Az analitikus megoldásban az ütközési pontban a relatív gyorsulásnak nemnegatívnak kell lennie, úgy ahogy a 4.3 fejezetben a relatív sebességnek. Az erők kiszámításához egy egyenlőtlenség-rendszert

kell megoldani, amely [2] szerint egy Quadratic programming probléma, ez nem került implementációra. A megoldáshoz egy $k \times k$ mátrixszal kéne számolni, a k az ütközési pontok száma, ez a mátrix sok ütközés esetén nagyon nagy lehet és a szimulációt jelentősen lassíthatja.

7.2. Iteratív megoldás

Az iteratív megoldás egy sokkal egyszerűbb elven alapszik: az 4.3 fejezetben leírtakat megismételjük egy párszor és reménykedünk, hogy egyik ütközési pontban sem akarnak a testek egymás felé mozogni. Bár ez a megoldás nem lesz soha tökéletes, elég jó közelítést ad a nyugalmi érintkezéshez is. Az közelítés pontosságát több módon lehet javítani, például ha az ütközési pontokat minden iterációban más sorrendben vizsgáljuk, vagy az első iterációkban nagyobb erőket engedünk, akkor a sebességek gyorsabban kiegyenlítődhetnek.

8. Fejezet

Fizikai könyvtár

8.1. Példa kód

A könyvtárban egy `World` típusú objektum reprezentálja a fizikai világot, ezt a következő módon hozhatjuk létre:

```
let mut world = WorldBuilder::new()
  .down_gravity(10.0) // 10 egység erősségű gravitáció, lefelé mutat
  .bounciness(0.9) // az ütközések nagyon rugalmasak
  .friction(0.1) // kicsi súrlódási tényező
  .build();
```

Az `add_staticbody` és `add_rigidbody` metódusok segítségével hozhatunk létre objektumokat:

```
world
  .add_staticbody(Shape::new_box(10.0, 1.0, 10.0))
  .pos(0.0, -5.0, 0.0)
  .finish();
let ball_id = world
  .add_rigidbody(Shape::new_sphere(1.0))
  .finish(); // RigidBodyId alapján lekérdezzük az objektumot
```

Az `update` metódus segítségével frissíthetjük a világot:

```
let time_step = 1.0 / 100.0;
let mut time = 0.0;
while time < 20.0 {
  time += time_step;
  world.update(time_step);
  let ball = world.get(ball_id).unwrap();
  if world.contacts().count() > 0 {
    println!(
      "floor hit at time {time:05.2}, new y momentum: {:.5}",
      ball.momentum().y
    );
  }
}
```


Kimenet:

```
floor hit at time 00.90, new y momentum: 4.26564
floor hit at time 02.54, new y momentum: 3.91772
floor hit at time 04.05, new y momentum: 3.63313
floor hit at time 05.45, new y momentum: 3.37763
floor hit at time 06.75, new y momentum: 3.13593
floor hit at time 07.95, new y momentum: 2.85678
floor hit at time 09.05, new y momentum: 2.63047
floor hit at time 10.07, new y momentum: 2.47649
floor hit at time 11.02, new y momentum: 2.26403
floor hit at time 11.90, new y momentum: 2.14356
floor hit at time 12.73, new y momentum: 2.01920
floor hit at time 13.51, new y momentum: 1.89271
floor hit at time 14.24, new y momentum: 1.76392
floor hit at time 14.92, new y momentum: 1.63219
floor hit at time 15.56, new y momentum: 1.57970
floor hit at time 16.17, new y momentum: 1.47846
floor hit at time 16.75, new y momentum: 1.43818
floor hit at time 17.30, new y momentum: 1.31296
floor hit at time 17.82, new y momentum: 1.30229
floor hit at time 18.32, new y momentum: 1.20360
floor hit at time 18.79, new y momentum: 1.15167
floor hit at time 19.24, new y momentum: 1.10437
floor hit at time 19.67, new y momentum: 1.04832
```

9. Fejezet

Eredmények

9.1. Gyorsítási módszerek összehasonlítása

A 6 fejezetben bemutatott módszerek közül választani kellett egyet a könyvtár implementációjában. A 1 táblázatban látható adatok szerint valós idejű szimulációkhoz a Sort-and-Sweep a legalkalmasabb algoritmus, hiszen nagy elemszámokra egyik algoritmus sem képes a megfelelő sebességgel szimulálni. Az eredmények ellenére a könyvtár jelenleg az OMT-t használja, mert valós idejű szimulációknál a különbség elhanyagolható, lassabb szimulációknál viszont jobban teljesít.

AABB-k száma	10	100	1000	10000	100000
Minden mindennel	102 ns	6.83 μ s	2100 μ s	252 ms	DNF
Sort-and-sweep	483 ns	7.04 μ s	261 μ s	19 ms	1830 ms
Sort-and-sweep 3	1611 ns	29 μ s	1829 μ s	210 ms	DNF
R-Tree	557 ns	39 μ s	941 μ s	21 ms	708 ms
R-Tree (split)	444 ns	24 μ s	771 μ s	14 ms	486 ms
OMT	468 ns	17 μ s	505 μ s	8.65 ms	164 ms
OMT (split)	455 ns	15 μ s	453 μ s	7.98 ms	146 ms

Táblázat 1: Gyorsítási módszerek összehasonlítása. A (split) sorokban az elemeket két struktúrába lettek elhelyezve és csak az elemek felével történt ütközés vizsgálat. A (split) eredmények közelebb vannak a valós helyzetekhez, ahol az elemek jelentős része nem mozoghat.

Függelék

A. OMT vágások kiszámítása

Az OMT a szinteket „felülről-lefelé” építi fel, de egy adott szinten a vágásokat „alulról-felfelé” számoljuk ki a következő módon:

```
fn calculate_splits(
    node_count: usize, // a létrehozandó csúcsok száma
    item_count: usize, // a szétosztandó elemek száma
    dimensions: usize, // a dimenziók száma
) -> Vec<Vec<usize>> {
    // vágások listája, minden vágás részei tartalmazzák,
    // hogy hány elemet tartalmaznak
    //
    // igazából csak dimensions vágásunk lesz,
    // de az algoritmusban a kezdőállapotot is egy vágásnak tekintjük
    let mut splits = Vec::with_capacity(dimensions + 1);
    // a kezdőállapotot hozzáadjuk a vágások listájához
    // a kezdő vágás minden része 1 elemet tartalmaz
    splits.push(vec![1; item_count]);
    for dim in 0..dimensions {
        // vesszük az előző vágást
        let last = splits.last().unwrap();
        // kiszámoljuk, hogy a jelenlegi dimenzió
        // szerint hány részre kell vágni
        let chunk_count = (node_count as f64)
            .powf((dimensions - dim) as f64 / dimensions as f64)
            .round() as usize;
        // az előző vágás nem garantáltan bontható egyenlő részekre,
        // ezért kiszámoljuk a kisebb és a nagyobb részek méretét
        let small_size = last.len() / chunk_count;
        let large_size = small_size + 1;
        // nagyobb részből annyi van,
        // ahány az előző részekből marad az egyenlőre osztás után
        let large_count = last.len() % chunk_count;
        let small_count = chunk_count - large_count;
        // számon tartja, hogy eddig hány részt használtunk
        let mut i = 0;
        let new_split = iter::empty()
            .chain(iter::repeat(small_size).take(small_count))
            .chain(iter::repeat(large_size).take(large_count))
            .map(|c| {
                // a last-ból c rész összevonásával kapjuk meg az új részt
                let res = last[i..][..c].iter().sum();
                i += c;
            })
            .collect();
        splits.push(new_split);
    }
    splits
```

```

        i += c;
        res
    })
    .collect();
    // az új vágást hozzáadjuk a vágásokhoz,
    // a következő iterációban ez lesz az utolsó vágás
    splits.push(new_split);
}
// a vágások listáját meg kell fordítani,
// mert alulról-felfele építettük fel,
// de felülről-lefelé fogjuk végrehajtani a vágásokat
splits.reverse();
// a kezdő vágást kitöröljük, mert arra nincs szükség
splits.pop();
splits
}

```

Irodalomjegyzék

- [1] David Baraff, „An Introduction to Physically Based Modeling: Rigid Body Simulation I—Unconstrained Rigid Body Dynamics”. [Online]. Elérhető: <https://www.cs.cmu.edu/~baraff/sigcourse/notesd1.pdf>
- [2] David Baraff, „An Introduction to Physically Based Modeling: Rigid Body Simulation II—Nonpenetration Constraints”. [Online]. Elérhető: <https://www.cs.cmu.edu/~baraff/sigcourse/notesd2.pdf>
- [3] Elmer G. Gilbert, Daniel W. Johnson, és S. Sathiya Keerthi, „A Fast Procedure for Computing the Distance Between Complex Objects in Three-Dimensional Space”. [Online]. Elérhető: <https://graphics.stanford.edu/courses/cs448b-00-winter/papers/gilbert.pdf>
- [4] William Bittle, „GJK (Gilbert–Johnson–Keerthi)”. [Online]. Elérhető: <https://dyn4j.org/2010/04/gjk-gilbert-johnson-keerthi/>
- [5] William Bittle, „EPA (Expanding Polytope Algorithm)”. [Online]. Elérhető: <https://dyn4j.org/2010/05/epa-expanding-polytope-algorithm/>
- [6] Norbert Beckmann, Hans-Peterbegel, Ralf Schneider, és Bernhard Seeger, „The R*-tree: An Efficient and Robust Access Method for Points and Rectangles+”. [Online]. Elérhető: <https://infolab.usc.edu/csci599/Fall2001/paper/rstar-tree.pdf>
- [7] Scott T. Leutenegger, Jeffrey M. Edgington, és Mario A. Lopez, „STR: A SIMPLE AND EFFICIENT ALGORITHM FOR R-TREE PACKING”. [Online]. Elérhető: <https://apps.dtic.mil/sti/pdfs/ADA324493.pdf>
- [8] Taewon Lee és Sukho Lee, „OMT: Overlap Minimizing Top-down Bulk Loading Algorithm for R-Tree”. [Online]. Elérhető: http://ftp.informatik.rwth-aachen.de/Publications/CEUR-WS/Vol-74/files/FORUM_18.pdf