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

2024-12-05

Tartalomjegyzék

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, 2024-12-05

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_unckeched`) 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 feljeszteni. 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 szögsebességből:

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

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

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:

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

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:

$$M = (x_J - x(t)) \times F, \quad \text{a forgatónyomaték}$$
$$\Delta L = M \cdot \Delta t = (x_F - x(t)) \times F \cdot \Delta t = (x_J - x(t)) \times J$$

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:

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

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:

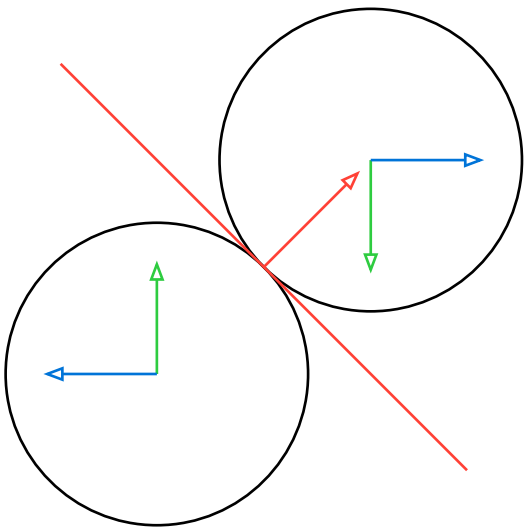
$$|\boldsymbol{c_1} - \boldsymbol{c_2}| < r_1 + r_2$$

Ha ütköznek, akkor az ütközési normál:

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

és az ütközési pontok:

$$\boldsymbol{p_1} = \boldsymbol{c_1} - r_1\boldsymbol{n}$$
$$\boldsymbol{p_2} = \boldsymbol{c_2} + r_2\boldsymbol{n}$$

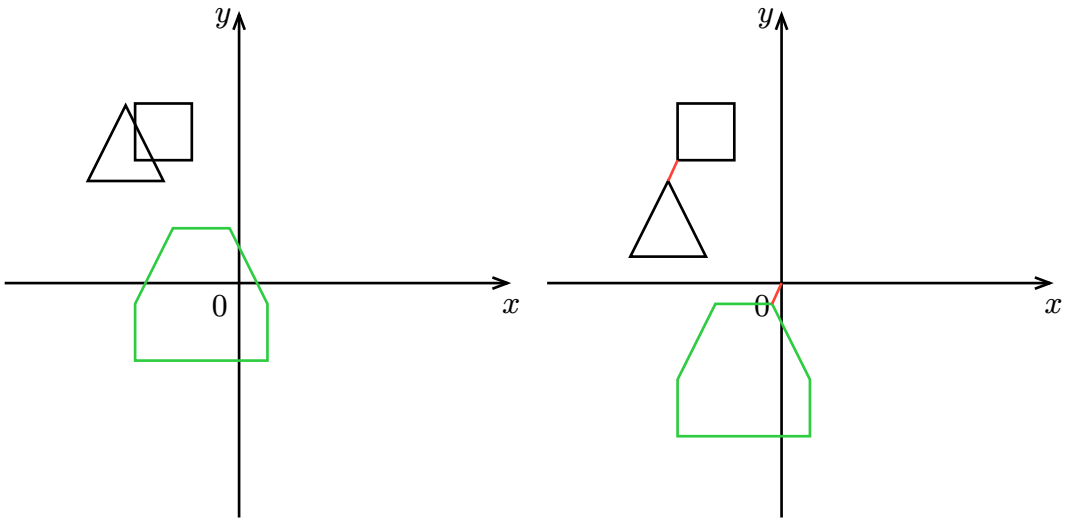


Ábra 1: Két gömb ütközik, piros az ütközési felület és normál, zöld az ütközés előtti sebességek, kék az ütközés utáni sebességek

5.2. Gilbert-Johnson-Keerthi algoritmus

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 sok időbe telne végig iterálni, de a különbséget felépítő szimplexeken (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:

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

Tehát az algoritmus a Minkowski különbség szimplexein iterál végig. Ezekkel a szimplexekkel egyre közelíteni szeretnénk az origót, míg vagy a szimplex tartalmazza az origót, vagy nem sikerült közelebb jutnunk az origóhoz. Az origóhoz úgy lehet közelíteni, hogy a szimplexnek vesszük az origóhoz a legközelebbi részszimplexét és a legközelebbi pontját, és a legközelebbi ponttal ellentétes irányba kérünk a support functiontől egy új pontot, amit hozzáadunk a szimplexhez.

Az algoritmus kétféleképpen került implementációra.

5.2.1. Baricentrikus koordinátákkal

A GJK célja találni egy szimplexet, ami tartalmazza az origót, ha egy adott szimplex nem tartalmazza az origót, akkor az origóhoz legközelebbi részszimplexét fogjuk bővíteni. Az egyik módszer az origó tartalmazás és a legközelebbi részszimplex kiszámításához baricentrikus koordinátákkal számol.

Egy n dimenziós szimplex pontjait a következő pontokkal határozunk meg: $S_1, S_2, ...S_n, S_{n+1}$

Egy P pontot a szimplex terén a következő módon írhatunk fel baricentrikus koordinátákkal:

$$P = \sum_{i=1}^{n+1} t_i \cdot A_i, \quad \sum_{i=1}^{n+1} t_i = 1$$

$$t_{n+1} = 1 - \sum_{i=1}^{n} t_i$$

$$P = \sum_{i=1}^{n} t_i \cdot A_i + \left(1 - \sum_{i=1}^{n} t_i\right) \cdot A_{n+1} = A_{n+1} + \sum_{i=1}^{n} t_i \cdot (A_i - A_{n+1})$$

$$\text{legyen } A'_i = A_i - A_{n+1}$$

$$P = A_{n+1} + \sum_{i=1}^{n} t_i \cdot A'_i$$

Ez a pont akkor és csak akkor van a szimplexben, ha

$$\bigvee_{i=1}^{n+1} t_i \geq 0$$

A P pont távolságának a négyzete az origótól:

$$d^2 = P^2 = A_{n+1}^2 + 2\sum_{i=1}^{n} t_i \cdot A_{n+1} \cdot A'_i + \sum_{i=1}^{n}\sum_{j=1}^{n} t_i \cdot t_j \cdot A'_i \cdot A'_j$$

Tudjuk, hogy ennek a távolságnak pontosan egy minimuma van, hiszen egy hipertérnek pontosan egy pontja lehet a legközelebb az origóhoz. A minimumot megkaphatjük a távolságnégyzet gradienséből:

$$\frac{\partial d^2}{\partial t_i} = 2A_{n+1} \cdot A'_i + 2\sum_{j=1}^{n} t_j \cdot A'_j \cdot A'_j = 0$$

$$\sum_{j=1}^{n} t_j \cdot A'_i \cdot A'_j = -A_{n+1} \cdot A'_i$$

Ezek az egyenletek egy $n+1$ változós lineáris egyenletrendszert alkotnak, amit a következő módon írhatunk fel mátrixokkal:

$$\begin{pmatrix} A'_1 \cdot A'_1 & A'_1 \cdot A'_2 & \cdots & A'_1 \cdot A'_n & 0 \\ A'_2 \cdot A'_1 & A'_2 \cdot A'_2 & \cdots & A'_2 \cdot A'_n & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ A'_n \cdot A'_1 & A'_n \cdot A'_2 & \cdots & A'_n \cdot A'_n & 0 \\ 1 & 1 & \cdots & 1 & 1 \end{pmatrix} \cdot \begin{pmatrix} t_1 \\ t_2 \\ \vdots \\ t_n \\ t_{n+1} \end{pmatrix} = \begin{pmatrix} -A_{n+1} \cdot A'_1 \\ -A_{n+1} \cdot A'_2 \\ \vdots \\ -A_{n+1} \cdot A'_n \\ 1 \end{pmatrix}$$

$$\begin{pmatrix} t_1 \\ t_2 \\ \vdots \\ t_n \\ t_{n+1} \end{pmatrix} = \begin{pmatrix} A'_1 \cdot A'_1 & A'_1 \cdot A'_2 & \cdots & A'_1 \cdot A'_n & 0 \\ A'_2 \cdot A'_1 & A'_2 \cdot A'_2 & \cdots & A'_2 \cdot A'_n & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ A'_n \cdot A'_1 & A'_n \cdot A'_2 & \cdots & A'_n \cdot A'_n & 0 \\ 1 & 1 & \cdots & 1 & 1 \end{pmatrix}^{-1} \cdot \begin{pmatrix} -A_{n+1} \cdot A'_1 \\ -A_{n+1} \cdot A'_2 \\ \vdots \\ -A_{n+1} \cdot A'_n \\ 1 \end{pmatrix}$$

Ha a mátrix nem invertálható (általában akkor fordul elő, ha a pontok nem alkottak szimplexet), akkor a szimplex minden részére rekurzívan futtatjuk az algoritmust. Ha tudjuk, hogy a szimplexnek az utolsó pontja a legújabban hozzáadott pont, akkor elég csak azt a pontot kitörölni és újrafuttatni.

Ha a kapott súlyokkal a pont a szimplexen kívülre esik, akkor a negatív súlyokhoz tartozó pontokat egyesével kivesszük a szimplexből és minden így kapott részre újra futtatjuk az algoritmust.

Az algoritmussal kapott szimplex az legközelebbi részszimplex az origóhoz. Ha a szimplex N dimenziós, akkor a szimplex tartalmazza az N dimenziós tér origóját, azaz befejezhetjük a GJK futtatását.

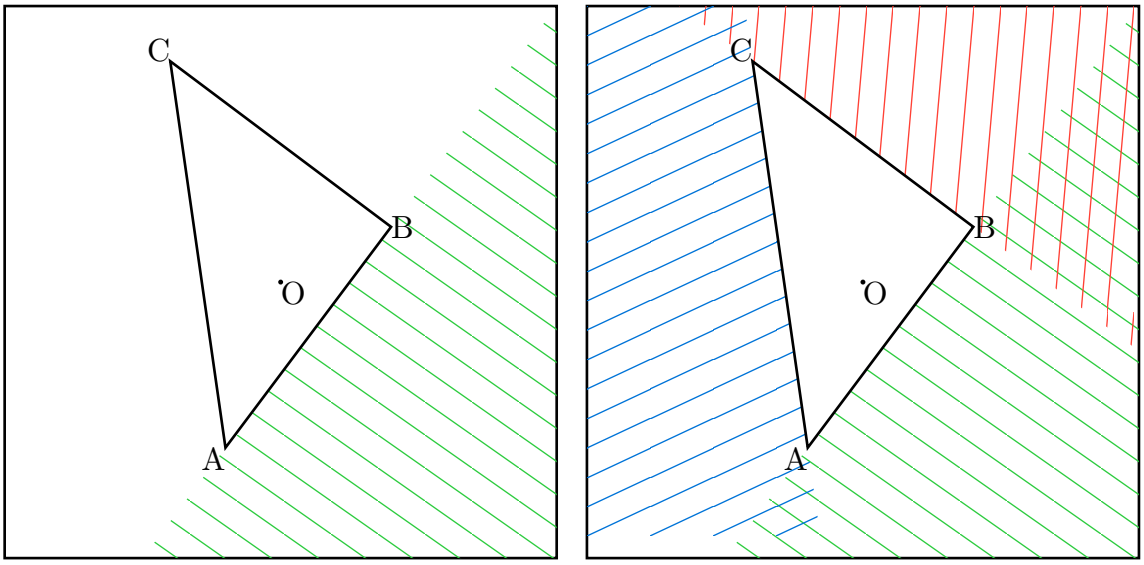
Ennek az algoritmusnak az előnye, hogy általános megoldást ad egy N dimenziós a legközelebbi részszimplex megtalására. Az algoritmus hátránya, hogy a mátrix szorzás és a szétágazó rekurzió miatt lassú.

5.2.2. Tér vágással

Ez az algoritmus a teret részszimplexek mentén félbe vágja és megnézi, hogy az origó az részszimplexen belüli vagy kívüli oldalra esik-e. A könyvtárban tetraéderekre lett implementálva, de az egyszerűség kedvéért itt csak háromszögekre lesz bemutatva.

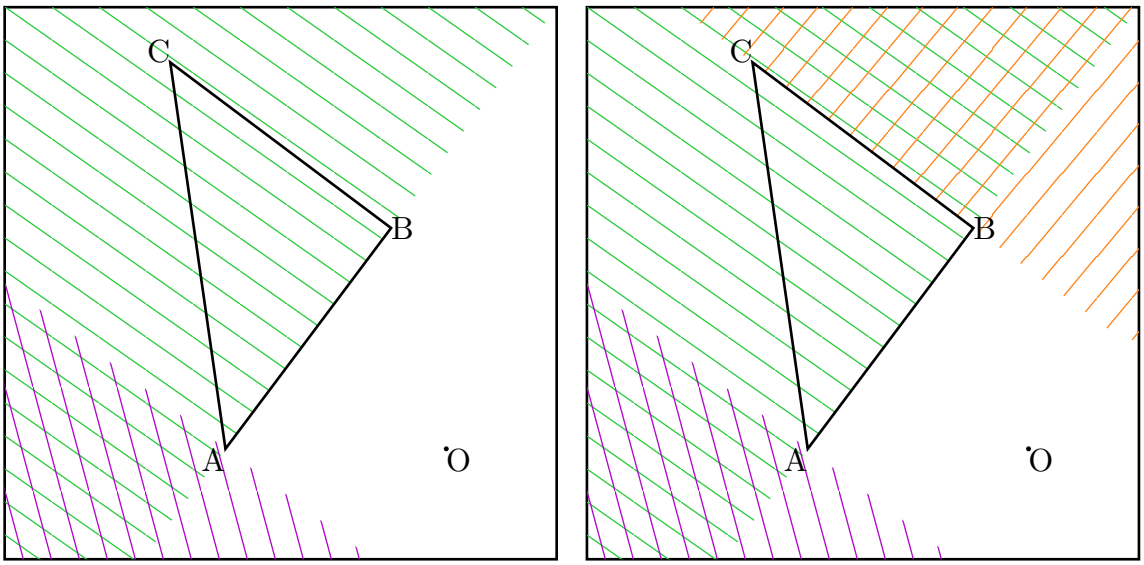
Az algoritmus menete:

- Háromszög vizsgálat:
  1. Vizsgáljuk meg, hogy az origó az AB szakasz melyik oldalán van. Ha az origó a C ponttól különböző oldalon van ($\overrightarrow{AO} \cdot \left[\overrightarrow{AB} \times \overrightarrow{AC} \times \overrightarrow{AB}\right] < 0$), akkor az origó biztosan nincs a háromszög területén, ugrás a *Szakasz vizsgálatra*
  2. Vizsgáljuk meg az origót és a BC szakaszt
  3. Vizsgáljuk meg az origót és a CA szakaszt
  4. Ha minden vizsgálat sikeres volt, akkor az origó a háromszögben van



Ábra 3: Az origó a háromszög közepén van. Az algoritmus 3 vágásból látta be.

- Szakasz vizsgálat : Tegyük fel, hogy az AB szakasz vizsgálatakor kiderült, hogy az origó a háromszög területén kívül esik.
  1. Vizsgáljuk meg, hogy az origó az A csúcs és az $\overrightarrow{BA}$ vektor által meghatározott síknak melyik oldalán van. Ha az origó nincs a B-vel egy oldalon ($\overrightarrow{AB} \cdot \overrightarrow{AO} < 0$), akkor az origó az A ponthoz van a legközelebb.
  2. Vizsgáljuk meg a B csúcs és az $\overrightarrow{AB}$ síkot az előzőhöz hasonlóan.
  3. Ha az origó mindkét síkon belül van, akkor az AB szakasz van az origóhoz a legközelebb.



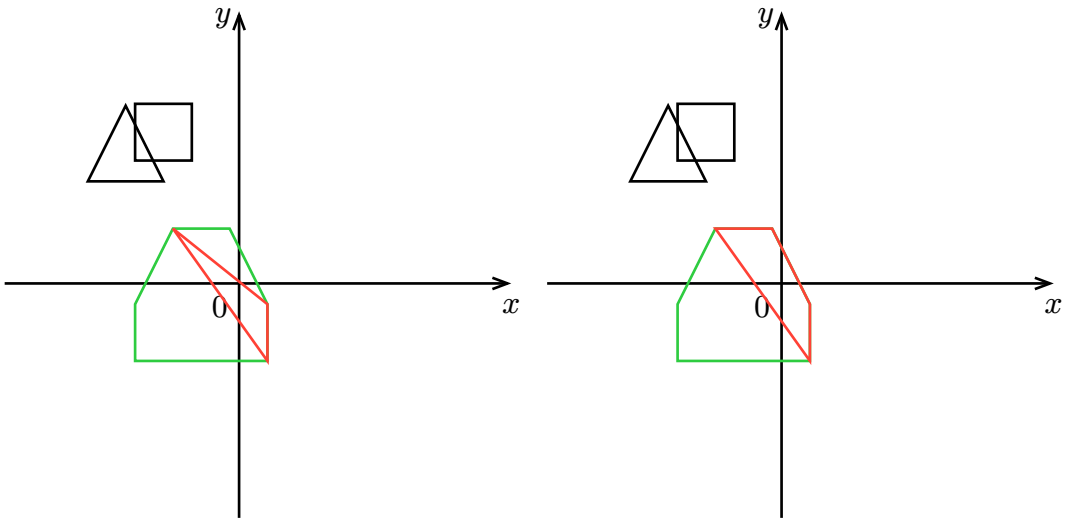
Ábra 4: Az origó az AB szakasz mellett van. Az algoritmus 3 vágásból látta be.

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

5.3. Expanding Polytope Algorithm

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 5: Bal oldal: az EPA egy lehetséges kezdő állapota. Jobb oldal: az EPA megtalálta az origóhoz legközelebbi oldalt, hiszen a legközelebbi oldal irányába nem tud messzebb menni.

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 [4] 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:

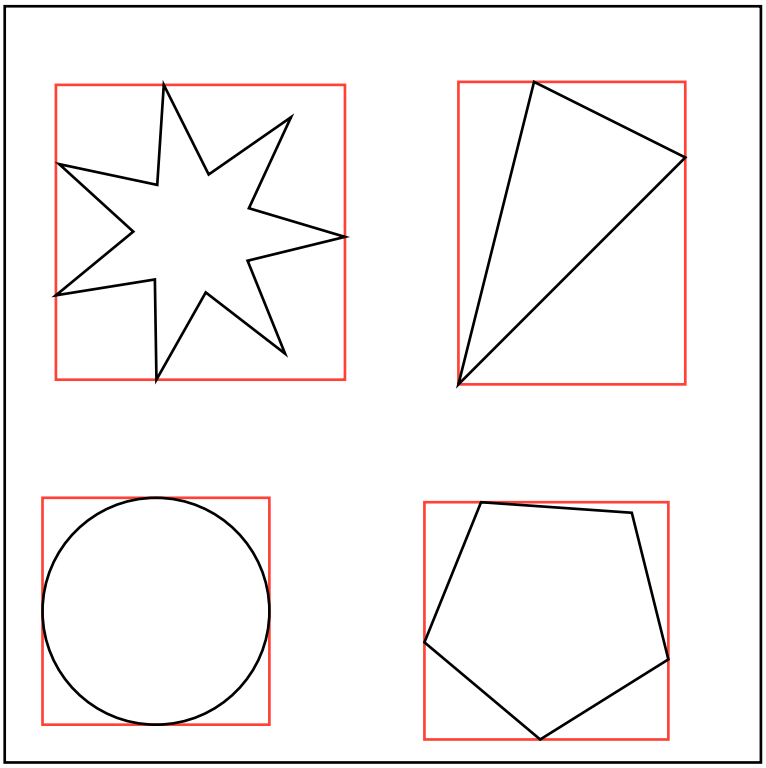
$$p' = p'_n - \frac{(p'_n - p'_0) \cdot \hat{n'}}{\hat{n} \cdot \hat{n'}} \cdot \hat{n}$$

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.

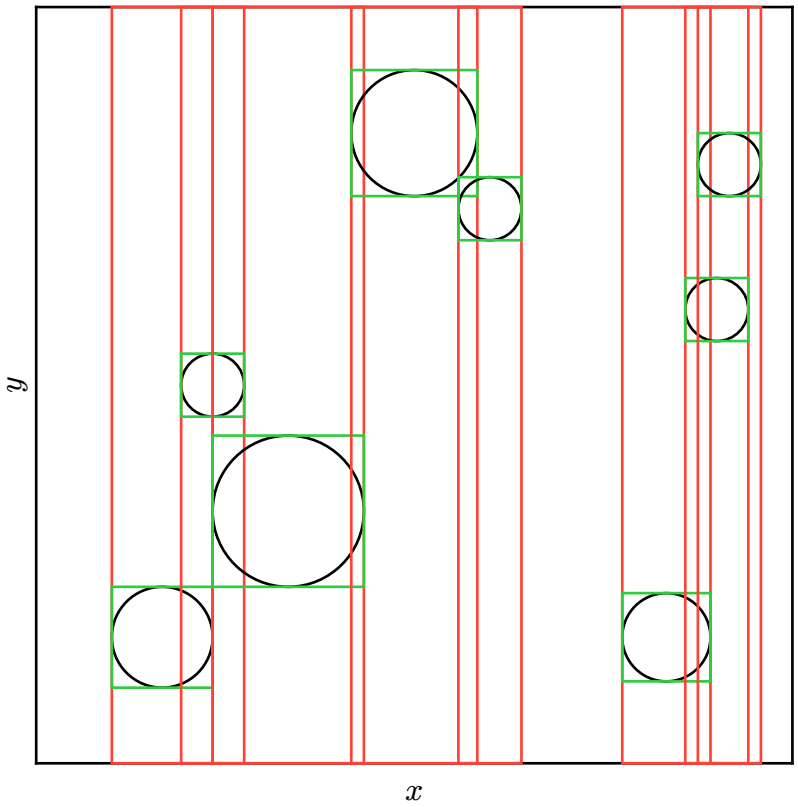


Ábra 6: Néhány különböző alaknak a minimális AABB-je.

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.

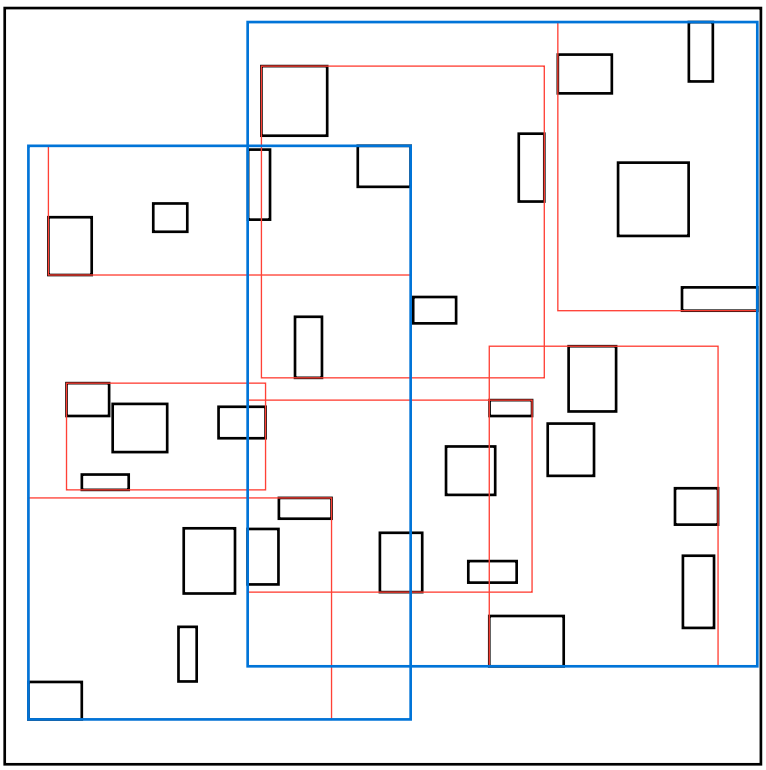


Ábra 7: A sort and sweep algoritmus intervallumai az x tengely szering. A jobb oldalon látható, hogy ha rossz tengelyt választunk, akkor nem segít sokat az algoritmus.

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 elemeit. Az R-Tree-be felépítésekor egyesével illesztjük be az AABB-ket. Az R-Tree-nék 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 [5] használ.



Ábra 8: Egy 2 szintű R-Tree. Látható, hogy néha átfedik egymást a csúcsok. Ezek az átfedések jelentősen csökkenthetik a lekérdezések sebességét.

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

Az R*-Tree [5] 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 [6] és az OMT [7] 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.

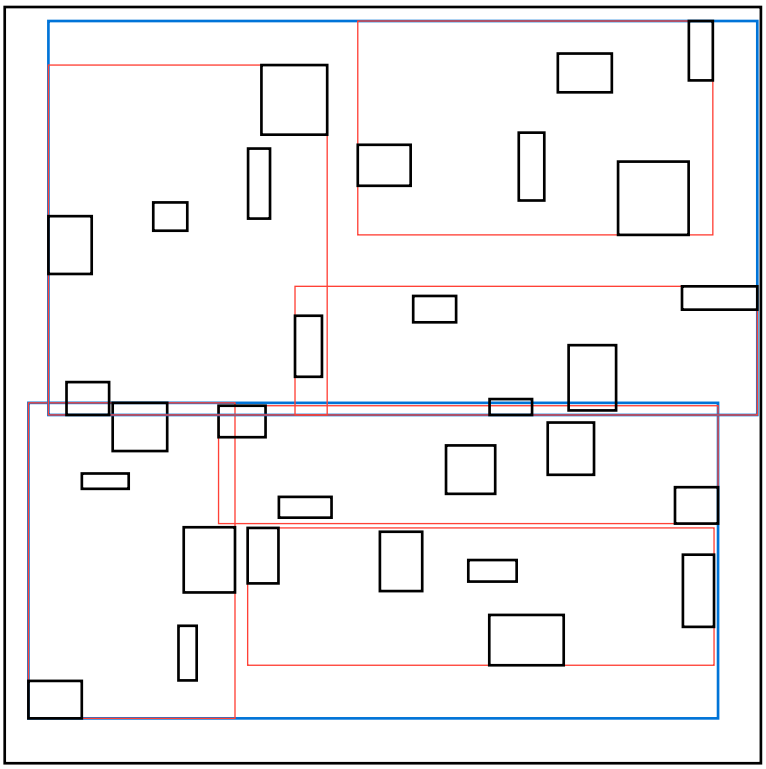
Ezzel a módszerrel hatékonyabb keresőfát tudunk felépíteni, ugyanis az algoritmus törekszik minimalizálni a részei között az átfedést.

6.3.1. Az algoritmus

Az algoritmus egy lépés ismétel addig, amíg a legalsó csúcshoz 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



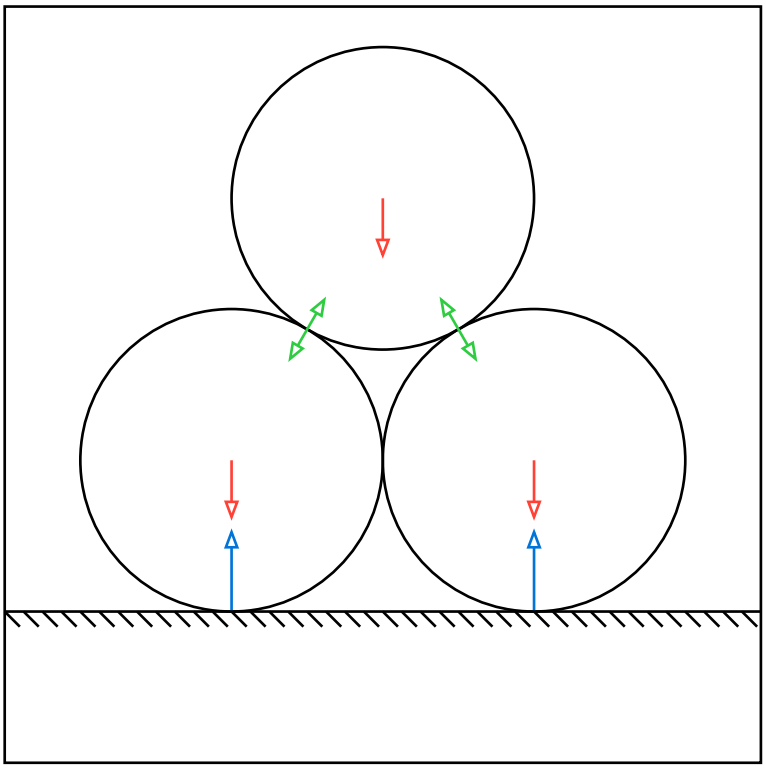
Ábra 9: Az Overlap Minimizing R-Tree. Látható, hogy kevesebb az átfedés a csúcsok között, mint az R-Treeben. Az OMT-ben a csúcsok kapacitásának kihasználtsága is jobb.

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 10: Nyugalmi érintkezéskor a testek normál irányú relatív sebessége 0. Erők viszont továbbra is hathatnak a testekre. Fontos hogy akkora erőket szimuláljunk az érintkező testek között, hogy ne gyorsuljanak egymás felé, de pusztán ezektől az erőktől nem válhatnak el a testek.

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. API dokumentáció

8.1.1. World

A szimuláció fő objektuma, ez tárolja a szimuláció allapotát és ez felelős az állapot frissítéséért.

8.1.1.1. Metódusok

`pub fn new(config: WorldConfig) -> Self`
Létrehoz egy új `World` objektumot, amely adott `WorldConfig` szerint van felkonfigurálva.

`pub fn get<'w, ID: ObjID<'w>>(&'w self, id: ID) -> Option<ID::Ref>`
Visszaadja az `ID`-hoz tartozó objektum referenciáját, ha létezik.

`pub fn add_staticbody(&mut self, shape: Shape) -> StaticbodyBuilder`
Visszaad egy `StaticbodyBuilder`-t, aminek segítségével létre tudunk hozni egy új statikus testet.

`pub fn staticbodies(&self) -> StaticbodyIter`
Visszaad egy iterátort a tárolt statikus testek referenciái felett.

`pub fn add_rigidbody(&mut self, shape: Shape) -> RigidbodyBuilder`
Visszaad egy `RigidbodyBuilder`-t, aminek segítségével létre tudunk hozni egy új merev testet.

`pub fn rigidbodies(&self) -> RigidbodyIter`
Visszaad egy iterátort a tárolt merev testek referenciái felett.

`pub fn contacts(&self) -> impl Iterator<Item = (Vec3, Vec3, Vec3)> + '_`
Visszaad egy iterátort az érintkezési pontok felett.

`pub fn update(&mut self, delta: Float)`
Lépteti a szimulációt `delta` idővel.

`pub fn aabbs(&self) -> impl Iterator<Item = (&AABB, usize)>`
Visszaad egy iterátort az AABB-k felett.

8.1.2. WoldConfig

Egy világ beállításait reperzentálja. Manuális létrehozás helyett a `WorldBuilder` használata ajánlott.

8.1.2.1. Mezők

`pub gravity: Vec3`
A gravitáció iránya.

`pub solver_steps: usize`
Az iteratív megoldó által megtett lépések száma.

`pub rb_separation_force: Float`
Az egymásba ragadt merev testek közötti taszító erő.

`pub sb_separation_force: Float`
Az egymásba ragadt merev testek és a statikus testek közötti taszító erő.

`pub bounciness: Float`
Az ütközések rugalmassági tényezője.

`pub friction: Float`
Az ütközések súrlódási tényezője.

8.1.3. WorldBuilder

Segít létrehozni egy új `World` objektumot.

8.1.3.1. Metódusok

`pub fn new() -> Self`
Létrehoz egy új `WorldBuilder` objektumot.

`pub fn no_gravity(self) -> Self`
Kikapcsolja a gravitációt.

`pub fn down_gravity(self, gravity: Float) -> Self`
Lefele mutató (y irányban negatív) gravitációt állít be.

`pub fn gravity(mut self, gravity: Vec3) -> Self`
Beállít egy tetszőleges gravitáció vektort.

`pub fn solver_steps(mut self, solver_steps: usize) -> Self`
Beállítja, hogy hány lépést tegyen meg az iteratív solver. Nem lehet 0.

`pub fn rb_separation_force(mut self, rb_separation_force: Float) -> Self`
Beállítja, hogy mekkora erővel lökjék el egymást az egymásba ragadt merev testek.

`pub fn sb_separation_force(mut self, sb_separation_force: Float) -> Self`
Beállítja, hogy mekkora erővel lökjék el egy statikus test a beleragadt merev testet.

`pub fn no_bounce(self) -> Self`
Tökéletesen rugalmatlan ütközést állít be.

`pub fn max_bounce(self) -> Self`
Tökéleteset rugalmas ütközést állít be.

`pub fn bounciness(mut self, bounciness: Float) -> Self`
Tetszőleges rugalmasságot állít be.

`pub fn no_friction(self) -> Self`
Kikapcsolja a súrlódást.

`pub fn friction(mut self, friction: Float) -> Self`
Tetszőleges súrlódási tényezőt állít be.

`pub fn build(self) -> World`
Létrehozza a felkonfigurált `World` objektumot.

8.1.4. RigidbodyRef

Egy merev test referenciát reprezentáló objektum. A `World` a merev testeket *struct of arrays* módon tárolja, ezért normális referneciákat nem lehet használni.

8.1.4.1. Metódusok

`pub fn shape(&self) -> &Shape`
Visszaadja a merev test alakját.

`pub fn inv_mass(&self) -> &Float`
Visszaadja a merev test tömegének az inverzét. A szimuláció ilyen formában tárolja a tömeget.

`pub fn mass(&self) -> Float`
Visszaadja a merev test tömegét.

`pub fn inverse_inertia(&self) -> &Mat3`
Visszaadja a merev test tehetetlenségi nyomatékának az inverzét. A tehetetlenségi nyomaték függ a test elfordulásától, ez az érték csak a testek léptetésekor frissül.

`pub fn position(&self) -> &Vec3`
Visszaadja a merev test pozícióját.

`pub fn rotation(&self) -> &Quat`
Visszaadja a merev test elfordulását.

`pub fn momentum(&self) -> &Vec3`
Visszaadja a merev test lendületét.

`pub fn angular_momentum(&self) -> &Vec3`
Visszaadja a merev test perdületét.

`pub fn local_velocity(&self, position: Vec3) -> Vec3`
Visszaadja a merev test lokális sebességét.

`pub fn impulse_effectivnes(&self, position: Vec3, direction: Vec3) -> Float`
Kiszámolja, hogy egy adott pontban egy adott irányú impulzus milyen hatékony-

sággal változtatná meg a lokális sebességet. Ez a test lokális tehetetlenségeként is elképzelhető.

8.1.5. StaticbodyRef

Egy statukis test referenciát reprezentáló objektum. A `World` a statikus testeket *struct of arrays* módon tárolja, ezért normális referenciákat nem lehet használni.

8.1.5.1. Metódusok

`pub fn shape(&self) -> &Shape`
Visszaadja a statikus test alakját.

`pub fn position(&self) -> &Vec3`
Visszaadja a statikus test pozícióját.

`pub fn rotation(&self) -> &Quat`
Visszaadja a statikus test elfordulását.

8.1.6. RigidbodyBuilder

Segít létrehozni egy új merev testet a szimulációban.

8.1.6.1. Metódusok

`pub fn new(world: &'w mut World, shape: Shape) -> Self`
Létrehoz egy új `RigidbodyBuilder`-t.

`pub fn mass(mut self, mass: Float) -> Self`
Beállítja a merev test tömegét.

`pub fn position(mut self, position: Vec3) -> Self`
Beállítja a merev test pozícióját.

`pub fn pos(self, x: Float, y: Float, z: Float) -> Self`
Beállítja a merev test pozícióját. Sokszor rövidebb három számot megadni, mint egy vektort.

`pub fn rotation(mut self, rotation: Quat) -> Self`
Beállítja a merev test elfordulását.

`pub fn finish(self) -> RigidbodyId`
Létrehoz egy merev testet a szimulációban. Ha ezt a metódust nem hívjuk meg, akkor nem jön létre a test, hiába használtuk a `World::add_rigidbody` metódust.

8.1.7. StaticbodyBuilder

Segít létrehozni egy új merev testet a szimulációban.

8.1.7.1. Metódusok

`pub fn new(world: &'w mut World, shape: Shape) -> Self`
Létrehoz egy új `StaticbodyBuilder`-t.

`pub fn position(mut self, position: Vec3) -> Self`
Beállítja a statikus test pozícióját.

`pub fn pos(self, x: Float, y: Float, z: Float) -> Self`
Beállítja a statikus test pozícióját. Sokszor rövidebb három számot megadni, mint egy vektort.

`pub fn rotation(mut self, rotation: Quat) -> Self`
Beállítja a statikus test elfordulását.

`pub fn finish(self) -> RigidbodyId`
Létrehoz egy statikus testet a szimulációban. Ha ezt a metódust nem hívjuk meg, akkor nem jön létre a test, hiába használtuk a `World::add_staticbody` metódust.

8.2. 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 lekérdezhetjük 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 µs	2100 µs	252 ms	DNF
Sort-and-sweep	483 ns	7.04 µs	261 µs	19 ms	1830 ms
Sort-and-sweep 3	1611 ns	29 µs	1829 µs	210 ms	DNF
R-Tree	557 ns	39 µs	941 µs	21 ms	708 ms
R-Tree (split)	444 ns	24 µs	771 µs	14 ms	486 ms
OMT	468 ns	17 µs	505 µs	8.65 ms	164 ms
OMT (split)	455 ns	15 µs	453 µs	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;
                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, „EPA (Expanding Polytope Algorithm)”. [Online]. Elérhető: https://dyn4j.org/2010/05/epa-expanding-polytope-algorithm/

[5] 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

[6] 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

[7] 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