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

2024-11-12

Tartalomjegyzék

Bevezetés **1**

1. Fizikai motorok **2**
1.1. Box2D 2
1.2. Rapier 2
1.3. PhysX 2

2. Rust **3**

3. Merevtest-szimuláció matematikája **4**
3.1. Lineáris mozgás 4
3.2. Forgás 4
3.3. Ütközésválasz 5
3.3.1. Impulzus és szögimpulzus 5
3.3.2. Lokális sebesség és lokális tehetetlenség 5
3.3.3. Normál irányú impulzus 6
3.3.4. Nem normál irányú (súrlódási) impulzus 6

4. Ütközés detektálás **7**
4.1. GJK 7
4.2. EPA 8
4.3. Ütközési pontok kiszámítása 9

5. Ütközés detektálás gyorsítása **10**
5.1. Sort-and-Sweep 10
5.2. R-Tree 11
5.3. OMT 11

6. Nyugalmi érintkezés **12**

7. Fizikai könyvtár **13**

8. Eredmények **14**

Irodalomjegyzék **15**

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, 2024-11-12

Szarkowicz Dániel
hallgató

Bevezetés

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

A merevtest-szimulációnak a célja a mindennapi fizikai objektumok egymásra hatásának szimulációja. Ehhez először fel kell építeni a fizikai modellt, ami szerint viselkedni fog a rendszer, majd be kell vezetni egy algoritmust, amely megmondja, hogy két test milyen kapcsolatban áll egymással. Végül optimalizálási algoritmusokkal kell gyorsítani a szimulációt, hogy akár több ezer testnek az interakcióit tudjuk egyszerre szimulálni.

1. Fejezet

Fizikai motorok

1.1. Box2D

2 Dimenziós fizikai motor C-ben

1.2. Rapier

2 és 3 Dimenziós fizikai motor Rust-ban

1.3. PhysX

valami valami nvidia

2. 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 indelxeléskor) 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.

3. Fejezet

Merevtest-szimuláció matematikája

3.1. 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 ütközés fejezetben fogok írni.

3.2. 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.

3.3. Ütközésválasz

3.3.1. 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:

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

3.3.2. 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:

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

3.3.3. 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]:

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

3.3.4. 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}-énekа 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.

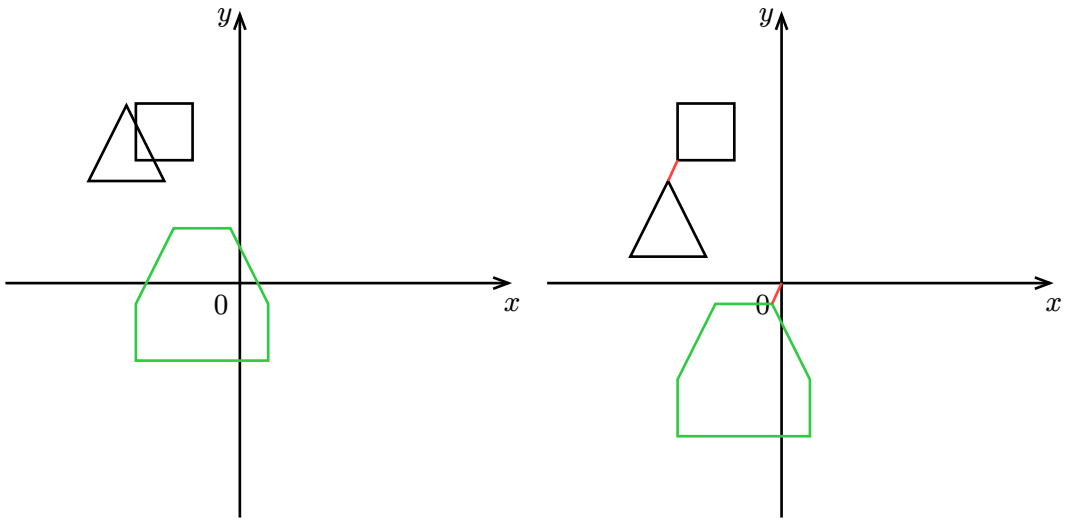
4. Fejezet

Ütközés detektálás

4.1. 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 1: 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 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.

Az első implementáció baricentrikus koordinátákkal kereste meg a legközelebbi részszimplexet és a legközelebbi pontot. Sajnos a legközelebbi részszimplex keresésnél néha nem egyértelmű, hogy melyik részszimplex 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 részszimplexenként két részre osztja és megnézi, hogy a részszimplexen belül vagy kívül esik-e az origó. Előbb vagy utóbb egy néhány részszimplex vagy körbe fogja az origót, és akkor tudjuk, hogy a részszimplexek 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 részszimplex) 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, innentől a sphere-collision fejezetben írt módon lehet kiszámolni, hogy a két test ütközik-e, és ha igen, akkor mik az ütközési paramétereik.

4.2. 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.

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.

4.3. Ütközési pontok kiszámítása

Polygon vetítés + clipping dyn4j-clipping

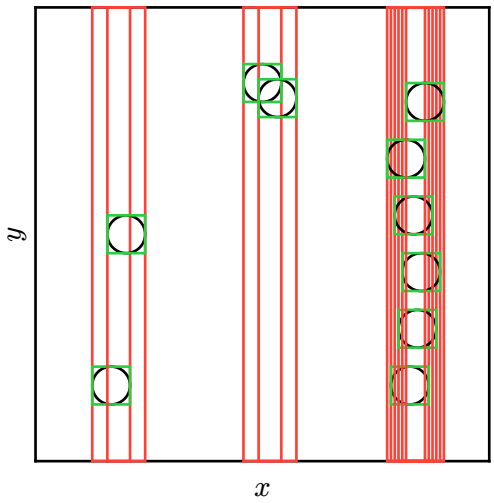
5. Fejezet

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.

5.1. 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 2: 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.

5.2. 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 [6] használ.

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.

5.3. OMT

OMT

6. Fejezet

Nyugalmi érintkezés

force solverek iteratív megoldás iterációk között átrendezés jobb minőségű propagációhoz

7. Fejezet

Fizikai könyvtár

kéne vmi world builder amivel be lehet állítani a paramétereit (gravitáció, solver steps, separation force)

```
struct World {}
```

```
struct RigidBodyRef {}
struct StaticBodyRef {}
```

8. Fejezet

Eredmények

ez régi, azóta mi van?: A program több, mint 1000 kockát képes valós időben szimulálni egy *AMD Ryzen 5 4500U* processzoron.

képek a többi demóról

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