Marko Huotari

T076KN

TÖRMÄYKSEN TUNNISTUS 3D-YMPÄRISTÖSSÄ

Opinnäytetyö
Tietotekniikka

Maaliskuu 2011

MIKKELIN AMMATTIKORKEAKOULU
Mikkeli University of Applied Sciences

KUVAILULEHTI

MIKKELIN AMMATTIKORKEAKOULU Mikkeli University of Applied Sciences	**Opinnäytetyön päivämäärä**
Tekijä(t) Marko Huotari	**Koulutusohjelma ja suuntautuminen** Tietotekniikka, ohjelmistokehitys

Nimeke

Törmäyksen tunnistus 3D-ympäristössä

Tiivistelmä

Törmäyksen tunnistus on laaja-alue, joka on keskeisessä osassa mm. peliohjelmoinnissa, animaatiossa, fysikaalisessa simuloinnissa ja robotiikassa. Peliohjelmoinnissa törmäyksen tunnistuksella luodaan vaikutelma reaalimaailmasta, esim. henkilö ei voi kävellä seinien läpi. Tämä työ keskittyi tutkimaan kuinka kahden kappaleen törmäys havaitaan 3D-ympäristössä.

Törmäyksen tunnistaminen on usean ongelman kokonaisuus esim. malleissa saattaa olla kymmeniä tuhansia polygoneja, jolloin kappaleiden törmäyksen tunnistus polygoneista laskettuna on yleensä laskennallisesti mahdotonta. Yleinen käytäntö on rakentaa mallin ympärille peite, jolle törmäyksen tunnistus tehdään. Ympäröivä peite on jokin tunnettu geometrinen kappale, joka sisältää kaikki mallin pisteet esim. pallo, laatikko tms. Ympäröivät peitteet ovat olennainen osa törmäyksen tunnistusta ja tästä syystä tämä työ jakaantui kahteen osaan. Ensimmäinen osa käsittelee ympäröiviä peitteitä ja toinen osa törmäystapahtuman tunnistusta.

Ympäröivistä peitteistä käsiteltiin orientoitunut laatikko ja konveksi peite. Orientoitunut laatikko on laatikko, jonka orientaatio kuvaa mallin tai pistejoukon orientaatiota, jonka ympärille se on määrätty. Orientoitunut laatikko määrättiin tässä työssä pääkomponentti analyysin avulla. Konveksi peite on pienin mahdollinen konveksi kappale joka voidaan määrätä pistejoukon ympärille. Tässä työssä on esitetty algoritmi konveksi peitteen määräämiseen.

Törmäyksen tunnistus keskittyi tässä työssä ongelmaan, jos käytössä on useita erilaisia ympäröiviä peitteitä esim. pallo, laatikko tms. tarvitaanko jokaiseen tilanteeseen oma algoritmi vai onko olemassa algoritmeja, jotka havaitsevat törmäyksen erilaisilla kappaleilla. Algoritmit jotka tässä työssä esiteltiin, ovat Gilber-Johnson-Keerthi (GJK) ja Chung-Wang. Molemmat algoritmit havaitsevat törmäyksen mielivaltaisilla konvekseilla kappaleilla ja molemmat perustuvat Minkowskin erotukseen.

Asiasanat (avainsanat)

Ympäröivät peitteet, Törmäyksen tunnistus, Orientoitunut laatikko, Konveksi peite, Gilbert-Johnson-Keerthi, GJK, Chung-Wang

Sivumäärä	**Kieli**	**URN**
47	Suomi	

Huomautus (huomautukset liitteistä)

Ohjaavan opettajan nimi	**Opinnäytetyön toimeksiantaja**
Selin, Jukka	MAMK

DESCRIPTION

MIKKELIN AMMATTIKORKEAKOULU Mikkeli University of Applied Sciences	**Date of the bachelor's thesis**
Author(s) Marko Huotari	**Degree programme and option** Information technology

Name of the bachelor's thesis

Collision Detection in 3D-environment

Abstract

Collision detection is a wide topic that is essential in many applications such as game programming, animation, physical simulation and robotics. In game programming collision detection creates the impression of real world e.g. people cannot walk through walls. This bachelor's thesis focused on how to detect an intersection between two objects in a 3D-environment.

Collision detection is a collection of many challenges e.g. models can have thousands of polygons, so detecting intersection from polygons is usually too costly. The common method is to construct a bounding volume around the model that is used for intersection detection. A bounding volume is some known geometrical shape that encloses all the points and polygons of the model e.g. sphere or a box. Bounding volumes are an essential part of intersection detection and that is why this study was split to two parts the first part focused on bounding volumes and the second part focused on intersection detection.

The bounding volumes covered in this study were oriented bounding box (OBB) and convex hull. An OBB is a box that is oriented the same way as the point set or model it encloses. In this study the OBB was computed using the principal component analysis (PCA). A convex hull is a minimum convex object that can be constructed around a point set. In this study an algorithm that computes the convex hull was described.

The intersection detection of this study focused on the question that if different types of bounding volumes were used e.g. spheres, boxes or convex hulls is a separate algorithm needed for each case or are there any algorithms that can detect intersection from different types of shapes. The algorithms covered in this study were Gilbert-Johnson-Keerthi (GJK) and Chung-Wang. Both of these algorithms are based on the Minkowski difference and can detect intersection between two arbitrary convex objects.

Subject headings, (keywords)

Bounding volumes, Collision detection, Intersection detection, Oriented bounding box, OBB, Convex hull, Gilbert-Johnson-Keerthi, GJK, Chung-Wang

Pages	**Language**	**URN**
47	Finnish	

Remarks, notes on appendices

Tutor	**Bachelor's thesis assigned by**
Selin, Jukka	MAMK

SISÄLTÖ

LIITTEET

1 JOHDANTO

Törmäyksen tunnistus on laaja aihealue, joka on keskeisessä osassa mm. peliohjelmoinnissa, animaatiossa, fysikaalisessa simuloinnissa ja robotiikassa. Törmäyksen tunnistus voidaan ajatella geometrisena ongelmana esim. mikä on kappaleiden välinen etäisyys, mitkä ovat lähimmät pisteet kappaleissa toisiinsa, kuinka välttää törmäys, reitinetsintä jne. Tässä työssä keskitytään tutkimaan 3D-peliohjelmoinnin näkökulmasta, törmääkö kaksi kappaletta. Peliohjelmoinnissa törmäyksen tunnistuksella luodaan vaikutelma reaalimaailmasta, esim. henkilö ei voi kävellä seinien läpi.

Törmäyksen tunnistaminen on usean ongelman kokonaisuus. Esimerkiksi malleissa saattaa olla kymmeniä tuhansia polygoneja, joten kappaleiden törmäyksen tunnistus polygoneista laskettuna on yleensä laskennallisesti mahdotonta. Yleinen käytäntö on rakentaa mallin ympärille peite, jolle törmäyksen tunnistus tehdään. Ympäröivä peite on jokin tunnettu geometrinen kappale, joka sisältää kaikki mallin pisteet esim. pallo, laatikko tms. Tästä syystä tämän työn alkuosa käsittelee ympäröiviä peitteitä. Ympäröivistä peitteistä käsitellään orientoitunut laatikko ja konveksi peite.

Oma ongelmansa on myös itse törmäystapahtuman tunnistaminen. Ongelma mihin tässä keskitytään, liittyy siihen, että jos käytössä on useita erilaista ympäröiviä peitteitä, tarvitaanko jokaiselle vaihtoehdolle oma algoritmi, vai onko olemassa algoritmeja, jotka havaitsevat törmäyksen erilaisilla kappaleilla.

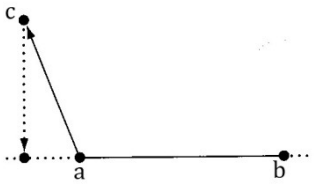
Työ on jäsennelty seuraavasti: Luvussa 2 käyn läpi kuinka janasta, tasosta ja kolmiosta lasketaan lähin piste mielivaltaiseen pisteeseen. Seuraavat kaksi lukua käsittelee kuinka orientoitunut laatikko ja konveksi peite määrätään. Luvussa 5 tutkitaan kuinka kahden paikallaan olevan konveksin kappaleen törmäys havaitaan Gilbert-Johnson-Keerthi ja Chung-Wang-algoritmeilla ja lopuksi luvussa 6 käsitellään liikkeessä olevia kappaleita. Kaikki tässä työssä esitellyt algoritmit on ohjelmoitu XNA-ympäristössä ja ne ovat liitteinä.

Matriiseja merkitään isoilla lihavilla kirjaimilla esim. **M** ja vektoreita merkitään pienillä lihavilla kirjaimilla esim. **v**. Merkintä **ab** tarkoittaa vektorien **a**:n ja **b**:n välistä vektoria (**b** – **a**) ja merkintä **abc** tarkoittaa pisteiden **a**, **b** ja **c** muodostaman pinnan normaalia.

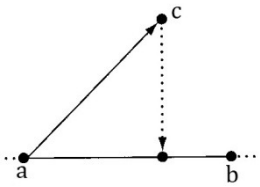
2 LÄHIN PISTE JANASTA, TASOSTA JA KOLMIOSTA

Tässä luvussa esitellään laskutavat lähimmän pisteen laskemiseksi janasta, tasosta ja kolmiosta mielivaltaiseen pisteeseen. Luku perustuu Ericsonin kirjaan [1].

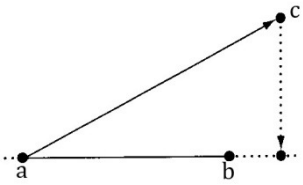
Lähin janan piste avaruuden pisteeseen c



KUVA 1. $t < 0$ [1]



KUVA 2. $0 \leq t \leq 1$ [1]



KUVA 3. $t > 1$ [1]

Jokainen piste janalta **ab** voidaan esittää funktiolla $P(t) = \mathbf{a} + t(\mathbf{b} - \mathbf{a})$, missä $0 \leq t \leq 1$. Lasketaan pisteen **c** projektio t janaan **ab** seuraavalla tavalla: $t = \mathbf{ac} \bullet \mathbf{ab} \,/\, (\mathbf{ab} \bullet \mathbf{ab})$, missä $\mathbf{ab} = \mathbf{b} - \mathbf{a}$ ja $\mathbf{ac} = \mathbf{c} - \mathbf{a}$.

1. Jos $t \leq 0$, niin lähin piste janasta **ab** pisteeseen **c** on piste **a** (kuva 1).
2. Jos $t \geq 1$, niin lähin piste on **b** (kuva 3).
3. Muuten, lähin piste on $\mathbf{a} + t(\mathbf{b} - \mathbf{a})$ (kuva 2) [1, s. 127 - 128.]

Lähin tason piste avaruuden pisteeseen q

Kaikki pisteet X tasosta π, toteuttaa yhtälön

$$\mathbf{n} \bullet (\mathrm{X} - \mathbf{p}) = 0 \qquad (1)$$

jossa **n** on tason normaali ja **p** on jokin piste tasosta. Merkitään pisteen **q** lähintä pistettä tasossa **r**:llä. Piste **q** voidaan myös ilmaista muodossa

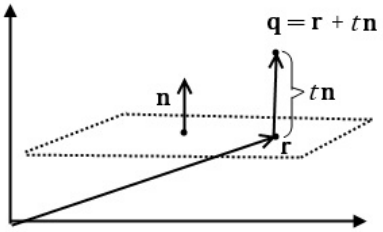
$$\mathbf{q} = t\mathbf{n} + \mathbf{r} \qquad (2)$$

jossa t on pisteen **q** etäisyys tasosta vektorin **n** pituuden yksiköissä. Sijoittamalla $\mathbf{r} = \mathbf{q} - t\mathbf{n}$ tason yhtälöön ja ratkaisemalla t, saadaan

$t = \mathbf{n} \bullet (\mathbf{q} - \mathbf{p}) / (\mathbf{n} \bullet \mathbf{n})$, jos $\mathbf{n}$ on normalisoitu $t = \mathbf{n} \bullet (\mathbf{q} - \mathbf{p})$ (3)

Lähin piste $\mathbf{r}$ tasosta saadaan sijoittamalla t yhtälöön $\mathbf{r} = \mathbf{q} - t\mathbf{n}$, josta saadaan

$$\mathbf{r} = \mathbf{q} - (\mathbf{n} \bullet (\mathbf{q} - \mathbf{p}) / (\mathbf{n} \bullet \mathbf{n}))\, \mathbf{n} \quad (4)$$

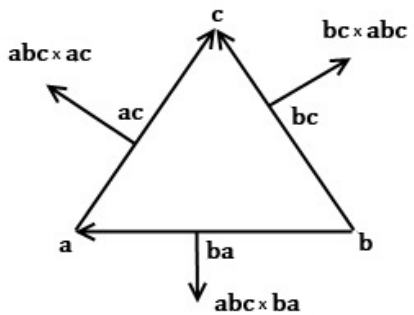


KUVA 4. Lähin piste tasosta

Lähin kolmion piste avaruuden pisteeseen p

Lähin piste kolmiolta, jonka kärkipisteet vastapäivään lueteltuna ovat **a**, **b** ja **c** pisteeseen **p** voidaan laskea mm. seuraavalla tavalla. Lasketaan kolmion muodostamasta tasosta lähin piste **q** pisteeseen **p**. Seuraavaksi tarkistetaan onko **q** kolmion ulkopuolella. Jos **q** on kolmion ulkopuolella, lasketaan lähin piste sivusta, joka on sitä lähinnä. Lasketaan ensin kolmion pinnan normaali $\mathbf{abc} = \mathbf{bc} \times \mathbf{ba}$.

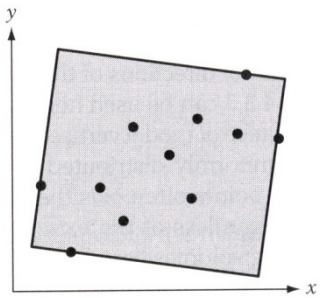
1. Jos $(\mathbf{bc} \times \mathbf{abc}) \bullet (\mathbf{q} - \mathbf{b}) \geq 0$, niin **q** on kolmion ulkopuolella ja lähin sivu on **bc**, joten lasketaan lähin piste sivusta **bc** pisteeseen **q**.
2. Jos $(\mathbf{abc} \times \mathbf{ba}) \bullet (\mathbf{q} - \mathbf{b}) \geq 0$, niin lähin piste on sivusta **ba**.
3. Jos $(\mathbf{abc} \times \mathbf{ac}) \bullet (\mathbf{q} - \mathbf{a}) \geq 0$, niin lähin piste on sivusta **ac**.
4. Jos mikään näistä mainituista ei toteutunut, niin **q** on lähin piste kolmiolta pisteeseen **p**.



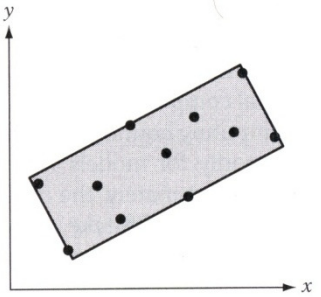
KUVA 5. **Lähin piste kolmiosta pisteeseen**

3 ORIENTOITUNEEN LAATIKON MÄÄRÄÄMINEN

Törmäyksen tunnistuksen helpottamiseksi voidaan pistejoukon ympärille määrätä mahdollisimman pieni laatikko. Tämän laatikon tulisi kuvata pistejoukon orientaatiota. Tällöin puhutaan orientoituneesta laatikosta. Orientoituneiden laatikoiden ongelma on, että hyvin ja huonosti lasketun laatikon kokoerot voivat olla erittäin suuria, kuten kuvista 6 ja 7 huomataan. Orientoituneen laatikon määrääminen on yllättävän monimutkaista ja tässä työssä se määrätään pääkomponenttianalyysin avulla. Laatikko voidaan esittää usealla tavalla: kahdeksalla kulmapisteellä, kolmella vektorilla ja keskipisteellä tai kuudella tasolla [3, s 101]. Tässä työssä laatikko esitetään kahdeksalla kulmapisteellä.



KUVA 6. Huonosti orientoitunut laatikko



KUVA 7. Hyvin orientoitunut laatikko

3.1 Pääkomponenttianalyysi

Pääkomponenttianalyysissä lasketaan pisteistä saadun kovarianssimatriisin ominaisvektorit, jotka ilmaisevat pistejoukon hajontasuunnat. Pistejoukko siirretään ominaisvektorien muodostamaan koordinaatistoon, minkä jälkeen lasketaan laatikon kulmapisteet. Lopuksi laatikon kulmapisteet siirretään takaisin alkuperäiseen koordinaatistoon. Kovarianssimatriisin laskemiseksi tarvitaan keskipoikkeamamatriisi, joka saadaan kaavalla 5.

$$\mathbf{D} = \begin{bmatrix} x_1 - \bar{x} & \dots & x_n - \bar{x} \\ y_1 - \bar{y} & \dots & y_n - \bar{y} \\ z_1 - \bar{z} & \dots & z_n - \bar{z} \end{bmatrix} \tag{5}$$

Kaavassa 5 n on pisteiden lukumäärä ja $\bar{x}$, $\bar{y}$ ja $\bar{z}$ ovat vektorien koordinaattien keskiarvot. Kovarianssimatriisi saadaan keskipoikkeamamatriisista laskemalla

$$\mathbf{C} = \mathrm{Cov}(\mathrm{x},\mathrm{y},\mathrm{z}) = \begin{bmatrix} \mathrm{Cov}(\mathrm{x},\mathrm{x}) & \mathrm{Cov}(\mathrm{x},\mathrm{y}) & \mathrm{Cov}(\mathrm{x},\mathrm{z}) \\ \mathrm{Cov}(\mathrm{y},\mathrm{x}) & \mathrm{Cov}(\mathrm{y},\mathrm{y}) & \mathrm{Cov}(\mathrm{y},\mathrm{z}) \\ \mathrm{Cov}(\mathrm{z},\mathrm{x}) & \mathrm{Cov}(\mathrm{z},\mathrm{y}) & \mathrm{Cov}(\mathrm{z},\mathrm{z}) \end{bmatrix} = \mathbf{D}\mathbf{D}^{\mathrm{T}}\frac{1}{n} \quad (6)$$

Kovarianssimatriisin ominaisvektorien laskemiseksi tarvitaan kovarianssimatriisin ominaisarvot λ, jotka saadaan yhtälöstä 7.

$$\det(\mathbf{C} - \lambda\mathbf{I}) = 0 \quad (7)$$

Ominaisarvo on arvo, joka vähennettynä, jokaisesta matriisin päälävistäjän elementistä tuottaa matriisin, jonka determinantti on nolla. Ominaisarvoja on yhtä monta kuin neliömatriisin rivejä. Ominaisarvojen ratkaisemiseksi joudutaan 3×3-matriisissa ratkaisemaan kolmannen asteen yhtälö ja 2×2-matriisissa toisen asteen yhtälö. Esimerkki 1 esittää kuinka 2×2-matriisista ratkaistaan ominaisarvot. Esimerkissä 1 a = Cov(x, x), b = Cov(y, x), c = Cov(x, y) ja d = Cov(y, y).

Esimerkki 1. $\det(\mathbf{C} - \lambda\mathbf{I}) = \begin{vmatrix} a-\lambda & c \\ b & d-\lambda \end{vmatrix} = \lambda^2 - \lambda(a+d) + ad - cb$

Ominaisarvoa vastaava ominaisvektori **x** saadaan ratkaisemalla yhtälöryhmä

$$(\mathbf{C} - \lambda\mathbf{I})\mathbf{x} = \mathbf{0} \quad (8)$$

Koska ominaisvektorit eivät ole yksikäsitteisiä, vain yksikköominaisvektorit lasketaan. Seuraavassa esimerkissä on laskettu yksi ominaisvektori 2×2-matriisista. Esimerkistä 2 havaitaan, että voidaan ainoastaan laskea ominaisvektorien suunnat, joten ominaisvektori kerrottuna millä tahansa nollasta poikkeavalla reaaliluvulla tuottaa myös matriisin ominaisvektorin. Esimerkin 2 ratkaisu pätee jos $c \neq 0$.

Esimerkki 2. $\begin{bmatrix} a-\lambda_0 & c \\ b & d-\lambda_0 \end{bmatrix} \cdot \begin{bmatrix} x_0 \\ y_0 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$

$$(a-\lambda_0)x_0 + cy_0 = 0$$

$$bx_0 + (d-\lambda_0)y_0 = 0$$

$$\mathbf{v}_0 = \begin{bmatrix} 1 \\ (\lambda_0 - a)/c \end{bmatrix}$$

Kun ominaisvektorit on laskettu, siirretään pistejoukko ominaisvektorien muodostamaan koordinaatistoon. Koordinaatistomuunnos saadaan kaavalla 9.

$$\mathbf{X}_B = \mathbf{B}^T\mathbf{X} \text{ , missä } \mathbf{B} = [\mathbf{v}_0 \quad \mathbf{v}_1 \quad \mathbf{v}_2] \text{ ja } \mathbf{X} = [\mathbf{v}_0 \quad \dots \quad \mathbf{v}_n] \tag{9}$$

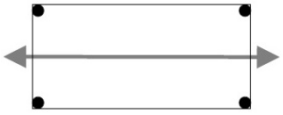
Kaavassa 9 **B** on yksikköominaisvektorit 3×3-matriisista, **X** on pistejoukko ja $\mathbf{X}_B$ on pistejoukko yksikköominaisvektorien muodostamassa koordinaatistossa. Koordinaatistomuunnoksen jälkeen lasketaan laatikon kaikki kahdeksan kulmapistettä $\mathbf{k}_0$ … $\mathbf{k}_7$ yksinkertaisesti etsimällä pisteet missä esiintyy suurimmat ja pienimmät x, y ja z arvot. Lopuksi siirretään kulmapisteet takaisin alkuperäiseen koordinaatistoon kaavalla 10.

$$\mathbf{X} = \mathbf{B}\mathbf{X}_B, \text{ missä } \mathbf{X} = [\mathbf{k}_0 \quad \dots \quad \mathbf{k}_7] \tag{10}$$

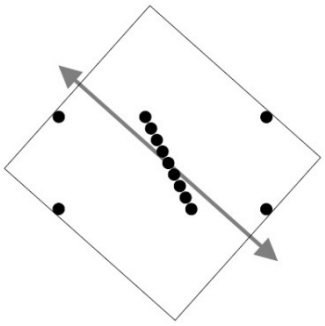
Kaavassa 10 $\mathbf{X}_B$ on laatikon kulmapisteet ominaisvektorien muodostamassa koordinaatistossa ja **X** on kulmapisteet alkuperäisessä koordinaatistossa.

3.2 Pääkomponenttianalyysimenetelmän ongelmat ja kehitys

Tämä luku perustuu Gottschalkin väitöskirjaan [2]. Pääkomponenttianalyysi toimii erittäin hyvin, kun pistejoukko on jakautunut tasaisesti. Mitä epätasaisemmin pistejoukko on jakautunut, sitä epätodennäköisemmin pienin mahdollinen laatikko on orientoitunut ominaisvektorien osoittamalla tavalla. Kuva 8 esittää tilannetta, jossa neljän tasaisesti jakautuneen pisteen ympärille on määrätty laatikko. Kuva 9 esittää tilannetta, jossa näiden neljän pisteen sisälle on lisätty joukko pisteitä. Kuvan 9 tilanteessa kuvan 8 laatikko olisi edelleen pienin mahdollinen laatikko, mutta sisälle lisätyt pisteet ovat vaikuttaneet ominaisvektorien osoittamiin hajontasuuntiin. Lisäämällä riittävä määrä pisteitä ääripisteiden sisälle, voidaan laatikon orientoitumista muuttaa lähes minkä muotoiseksi tahansa.

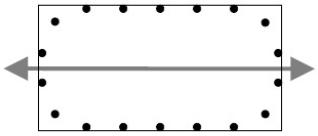


KUVA 8. Tasainen jakauma

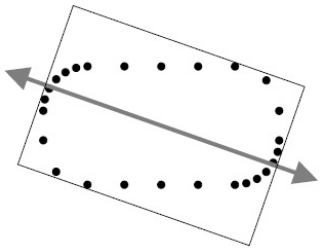


KUVA 9. Epätasainen jakauma

Tätä ongelmaa voidaan auttaa huomioimalla vain ääripisteet. Laskemalla ensin pistejoukon konveksi peite saadaan ääripisteet, joiden avulla määrätty laatikko on lähempänä pienintä mahdollista. Mutta tämäkään ei auta, jos ääripisteet ovat jakautuneet epätasaisesti. Kuva 10 esittää tilannetta, jossa ääripisteet ovat jakautuneet tasaisesti ja kuva 11 esittää tilannetta, jossa ääripisteet ovat jakautuneet epätasaisesti.



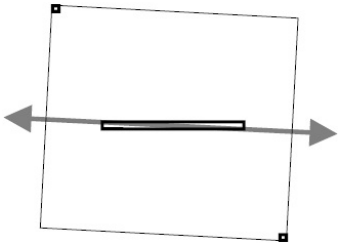
KUVA 10. Tasainen jakauma



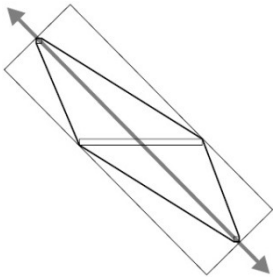
KUVA 11. Epätasainen jakauma

Tästä voidaan päätellä, että pelkästään pisteiden perusteella ei voi laskea pienintä mahdollista laatikkoa.

Joudutaan tutkimaan kuinka ääripisteet yhdistyvät toisiinsa. Näytteistämällä jokainen jana saadaan ominaisvektorit osoittamaan suuntiin, jotka ilmoittavat mallin todellisen orientaation. Tämäkään ei vielä ole riittävää, kun kuvitellaan kuvan 12 tilanne, jossa malli koostuu janasta ja kahdesta pisteestä. Nyt ominaisvektorit osoittavat mallin todelliseen hajontasuuntaan, mutta pienin mahdollinen laatikko ei ole orientoitunut näiden ominaisvektorien osoittamalla tavalla. Ratkaisu on laskea pistejoukon konveksi peite, josta jokainen särmä ja kolmio näytteistetään (kuva 13). Gottschalk ei kuitenkaan työssään kirjaimellisesti tehnyt näytteistystä vaan käytti tilastollisia menetelmiä. [2, s 41 – 46.]



KUVA 12. Jana ja kaksi pistettä

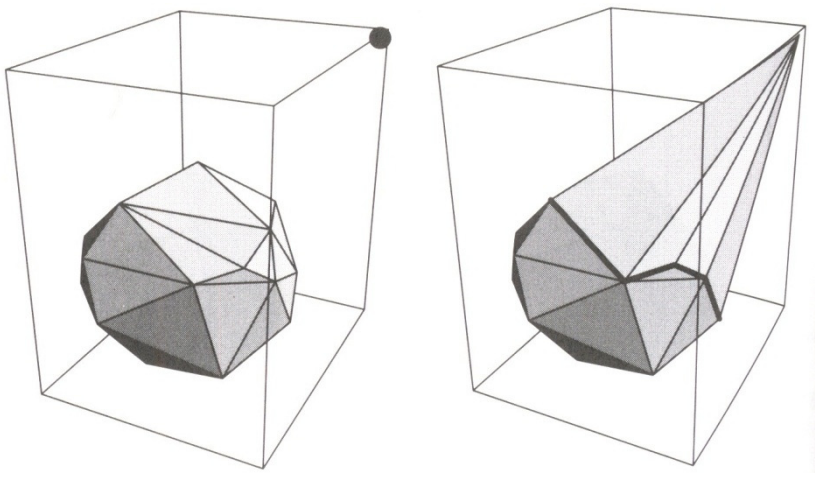


KUVA 13. Konveksi peite

4 KONVEKSI PEITTEEN MÄÄRÄÄMINEN

Tässä esitellään algoritmi konveksin peitteen laskemiseksi, jota O'Rourke [6, s. 115 - 140] kutsuu nimellä inkrementaalinen algoritmi. Algoritmi rakentaa pistejoukon ympärille konveksin peitteen, joka rakentuu kolmioista.

Algoritmin idea on seuraavanlainen. Rakennetaan alussa neljästä mielivaltaisesta pisteestä kartio. Otetaan piste kartion ulkopuolelta ja etsitään mitkä tahkot ovat näkyvissä tästä pisteestä katsottuna. Etsitään näkyvistä tahkoista uloimmat näkyvät särmät, kutsutaan näitä särmiä näkyvän alueen reunoiksi. Muodostetaan näistä särmistä uudet tahkot, jotka yhdistyvät pisteeseen ja poistetaan näkyvät tahkot. Seuraavaksi valitaan toinen piste, monitahokkaan ulkopuolella. Etsitään näkyvät tahkot. Etsitään näkyvistä tahkoista särmät, jotka kuuluvat näkyvän alueen reunoihin. Muodostetaan näistä särmistä ja lisättävästä pisteestä uudet tahkot ja poistetaan näkyvät tahkot. Tämä toistetaan jokaiselle pisteelle, kunnes ei ole enää pisteitä monitahokkaan ulkopuolella. Algoritmi tavallaan venyttää monitahokasta, kunnes kaikki pisteet ovat sen sisällä.



KUVA 14. Monitahokas ennen ja jälkeen uuden pisteen lisäystä [6]

On tärkeä huomioida se, että uudet tahkot rakennetaan vain särmistä jotka kuuluvat näkyvän alueen reunoihin, koska tarkoitus on rakentaa monitahokas, jonka tahkot sulkevat sisälleen kaikki pisteet. Se, mistä tiedetään, kuuluuko särmä näkyvän alueen reunoihin, voidaan päätellä siitä, että jos ainoastaan yksi siihen yhdistyvistä tahkoista on näkyvä, täytyy sen kuulua näkyvän alueen reunoihin. Kuvassa 14 tummennetut särmät esittävät näkyvän alueen reunoja.

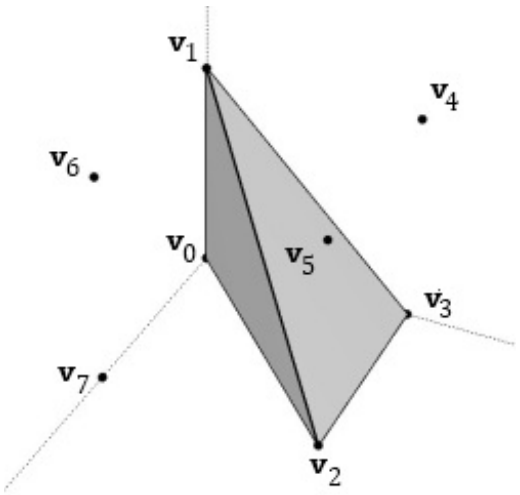
4.1 Näkyvien tahkojen selvittäminen pisteestä katsottuna

Näkyvät tahkot annetusta pisteestä katsottuna saadaan selville seuraavalla tavalla: Jos tahkon eli kolmion pisteet on kierretty vastapäivään, pisteestä katsottuna, on se näkyvä tahko. Tämä tietysti edellyttää sen, että monitahokas on rakennettu niin, että sen jokainen tahko (kolmio) on ulkoapäin katsottuna orientoitu vastapäivään. Kolmion orientaatio pisteestä katsottuna saadaan selville determinantin avulla. Kartion kärkipisteistä laskettu determinantti on itseisarvoltaan kuusi kertaa suurempi kuin kartion tilavuus (kaava 11). Determinantti on negatiivinen, jos kolmio on kierretty vastapäivään neljännestä pisteestä katsottuna, joten tällä tavalla saadaan selville tahkojen orientaatiot. Tämän algoritmin kannalta ei kaavassa 11 tarvitse suorittaa jakolaskua, vain etumerkillä on merkitystä.

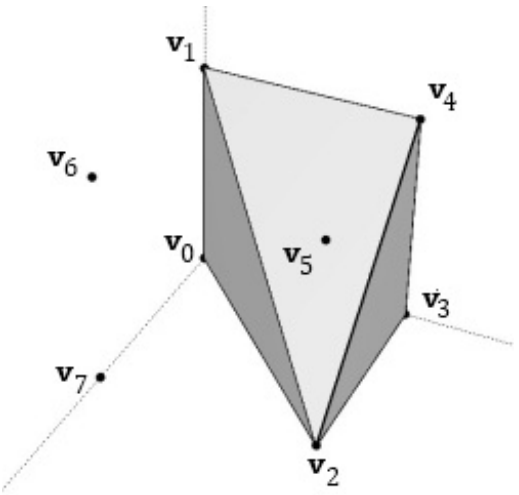
$$\mathrm{V} = \frac{1}{6}\begin{vmatrix} a_x & a_y & a_z & 1 \\ b_x & b_y & b_z & 1 \\ c_x & c_y & c_z & 1 \\ d_x & d_y & d_z & 1 \end{vmatrix} \tag{11}$$

4.2 Esimerkki konveksi peitteen määräämisestä

Tämä esimerkki esittää kuinka algoritmi muodostaa kahdeksan pisteen ympärille konveksi peitteen. Pistejoukko muodostaa kuution. Algoritmin alussa muodostetaan neljästä mielivaltaisesta pisteestä kartio (kuva 15). Tämän jälkeen valitaan mielivaltainen piste sen ulkopuolelta, tässä esimerkissä piste $\mathbf{v}_4$. Pisteestä $\mathbf{v}_4$ ainoa näkyvä tahko on pisteiden $\mathbf{v}_1$, $\mathbf{v}_2$ ja $\mathbf{v}_3$ muodostama tahko, joten luodaan kolme uutta tahkoa särmistä, jotka muodostavat näkyvän alueen reunat. Uudet tahkot muodostuvat pisteistä ($\mathbf{v}_3$, $\mathbf{v}_1$, $\mathbf{v}_4$), ($\mathbf{v}_2$, $\mathbf{v}_3$, $\mathbf{v}_4$) ja ($\mathbf{v}_1$, $\mathbf{v}_2$, $\mathbf{v}_4$). Lopuksi poistetaan näkyvä tahko. Kuva 16 esittää monitahokasta sen jälkeen kun piste $\mathbf{v_4}$ on lisätty.

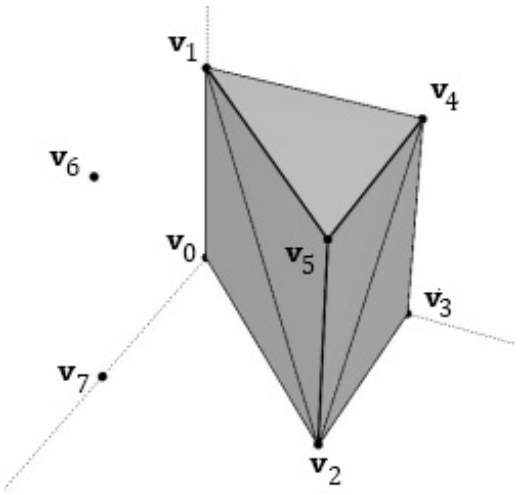


KUVA 15. Alkutilanne [6]

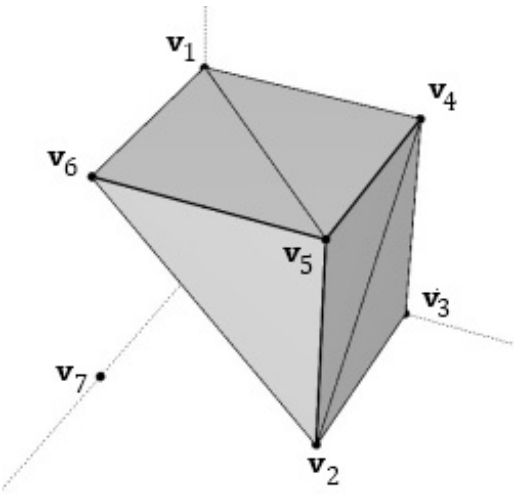


KUVA 16. Piste v_4 lisättynä [6]

Kuvassa 16 pisteestä $\mathbf{v}_5$ ainoa näkyvä tahko on pisteiden $\mathbf{v}_1$, $\mathbf{v}_2$ ja $\mathbf{v}_4$ muodostama tahko, joten luodaan kolme uutta tahkoa ($\mathbf{v}_1$, $\mathbf{v}_2$, $\mathbf{v}_5$), ($\mathbf{v}_1$, $\mathbf{v}_5$, $\mathbf{v}_4$) ja ($\mathbf{v}_5$, $\mathbf{v}_2$, $\mathbf{v}_4$) sekä poistetaan tahko ($\mathbf{v}_1$, $\mathbf{v}_2$, $\mathbf{v}_4$). Kuva 17 esittää monitahokasta sen jälkeen kun piste $\mathbf{v}_5$ on lisätty.

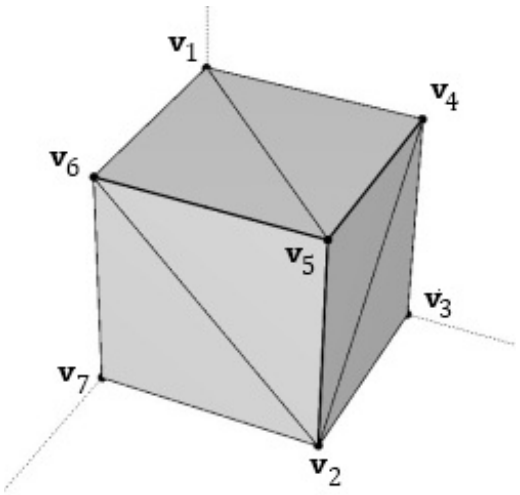


KUVA 17. Piste v_5 lisättynä [6]



KUVA 18. Piste v_6 lisättynä [6]

Kuvassa 17 pisteestä $\mathbf{v}_6$ on kaksi näkyvää tahkoa; tahkot ($\mathbf{v}_1$, $\mathbf{v}_2$, $\mathbf{v}_5$) ja ($\mathbf{v}_0$, $\mathbf{v}_2$, $\mathbf{v}_1$). Muodostetaan tahkot ($\mathbf{v}_6$, $\mathbf{v}_0$, $\mathbf{v}_1$), ($\mathbf{v}_6$, $\mathbf{v}_5$, $\mathbf{v}_1$), ($\mathbf{v}_6$, $\mathbf{v}_2$, $\mathbf{v}_5$), ($\mathbf{v}_6$, $\mathbf{v}_0$, $\mathbf{v}_2$) ja poistetaan näkyvät tahkot. Särmästä ($\mathbf{v}_1$, $\mathbf{v}_2$) ei muodosteta tahkoa, koska se yhdistyy kahteen näkyvään tahkoon, joten se ei kuulu näkyvän alueen reunoihin. Kuva 18 esittää monitahokasta sen jälkeen kun piste $\mathbf{v}_6$ on lisätty. Lopuksi lisätään piste $\mathbf{v}_7$ monitahokkaaseen ja saatu konveksi peite on esitetty kuvassa 19.



KUVA 19. Lopullinen konveksi peite [6]

4.3 Tietorakenteet ja algoritmi

Jotta pystytään tietämään kuinka jokainen kolmio on orientoitunut ja kuuluuko näkyvän tahkon särmä näkyvän alueen reunoihin, joudutaan ylläpitämään seuraavanlaisia tietorakenteita tahkon ja särmän kohdalla.

- Tahkolle tallennetaan tieto sen kärkipisteistä, särmistä sekä totuusarvomuuttuja, johon merkitään onko se näkyvä tahko.
- Särmälle tallennetaan tieto sen kärkipisteistä sekä mille tahkoille se kuuluu.
- Pisteelle tallennetaan x, y ja z koordinaattien lisäksi tieto särmästä, joka on apumuuttuja, ja jonka tarkoitus selviää myöhemmin.

Syy sille miksi kolmiolle asetetaan kärkipisteet erikseen, eikä katsota niitä särmistä on, että yksi särmä kuuluu aina kahteen tahkoon ja näiden orientaation pisteestä katsottuna, voi olla eri, joten kärkipisteiden avulla ylläpidetään tietoa kolmion orientaatiosta.

Pseudokoodi 1 konveksi peitteen määräämiseksi etenee seuraavasti: Merkitään jokaiselle pisteelle, mitkä tahkot ovat pisteestä näkyvissä laskemalla jokaiselle tahkolle, pisteen ja tahkon muodostaman kartion tilavuus kaavalla 11. Jos tahko on merkitty näkyväksi, tarkistetaan kuuluuko joku sen särmistä näkyvän alueen reunoihin. Koska särmän tietoihin on tallennettu mihin tahkoihin se yhdistyy, voidaan yksinkertaisesti katsoa onko niistä vain yksi merkitty näkyväksi. Mikäli näin on, särmä kuuluu näkyvän alueen reunoihin. Muodostetaan särmistä, jotka kuuluvat näkyvän alueen reunoihin, ja lisättävästä pisteestä uudet tahkot ja lisätään ne taulukkoon. Lopuksi poistetaan

näkyvä tahko kuului joku sen särmistä näkyvän alueen reunoihin tai ei. Kuten pseudokoodista huomaa, lopputuloksena on taulukko tahkoja, jotka muodostavat konveksi peitteen.

Pseudokoodi 1. Konveksi peite

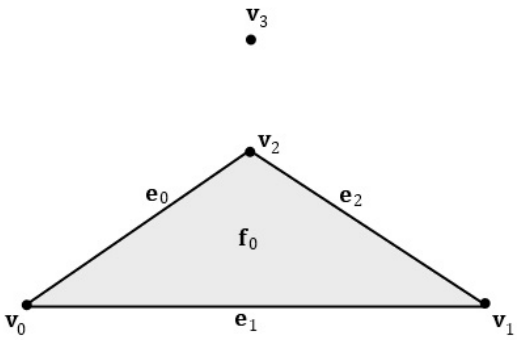
```
Muodosta kartio
Lisää kartion tahkot taulukkoon

for each piste
    Merkitse jokainen näkyvä tahko
    for each tahko
        if tahko on näkyvä
            for each särmä in tahko
                if ainoastaan yksi särmään yhdistyvistä tahkoista on näkyvä
                    Muodosta särmästä ja pisteestä tahko
                    Lisää luotu tahko taulukkoon
                end if
            end for
            Poista näkyvä tahko taulukosta
        end if
    end for
end for
```

Ainoa ero toteutuksessa ja pseudokoodissa on, että toteutuksessa ei ole aloitettu kartiosta vaan kahdesta kolmiosta, joissa on samat kärkipisteet mutta vastakkainen orientaatio. Kahdesta kolmiosta on aloitettu, koska alussa joudutaan manuaalisesti rakentamaan kappale, josta lähdetään rakentamaan konveksi peitettä ja kartio olisi vaatinut huomattavasti enemmän koodirivejä. Kahdesta vastakkain kierretystä kolmiosta voi aloittaa, koska heti ensimmäisen pisteen lisäyksen jälkeen kappale on kartio. On myös tärkeä, että kolmiot on vastakkain kierretty, koska muuten ensimmäisen pisteen lisäyksen jälkeen yksi tahko on orientoitu väärin. Alkutilanteessa rakennettu kappale täytyy myös rakentaa niin, että jokaisen särmän tietoihin on tallennettu mihin tahkoihin se yhdistyy. Toteutuksessa on myös tarkistus, että ne kolme pistettä joista rakennettiin kaksi päinvastoin kierrettyä kolmiota, todella muodostavat kolmion (katso liitteet 2 ja 5).

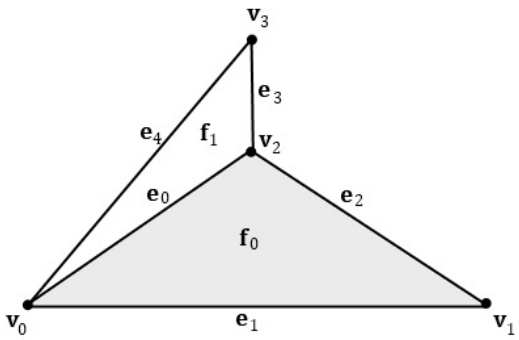
4.4 Uuden tahkon luonti

Uuden tahkon luonti on konveksi peitteen muodostamisessa hankalin osa. Uusi tahko täytyy luoda niin, että sen kärkipisteet on orientoitu vastapäivään. Lisäksi jokaisen särmän tietoihin on tallennettava mihin kahteen tahkoon se yhdistyy.

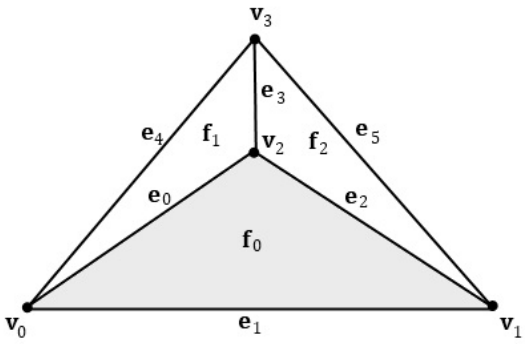
Kuvitellaan esimerkiksi kuvan 20 tilanne, jossa särmät $\mathbf{e}_0$, $\mathbf{e}_1$ ja $\mathbf{e}_2$ muodostavat näkyvän alueen reunat ja piste $\mathbf{v}_3$ aiotaan lisätä monitahokkaaseen. Luodaan särmästä $\mathbf{e}_0$ ja pisteestä $\mathbf{v}_3$ uusi tahko $\mathbf{f}_1$ (kuva 21). Lisätään pisteen $\mathbf{v}_2$ tietoihin, että siitä on luotu särmä $\mathbf{e}_3$ ja pisteen $\mathbf{v}_0$ tietoihin, että siitä on luotu särmä $\mathbf{e}_4$. Joka kerta kun pisteestä luodaan särmä ja siitä ei ole aiemmin luotu särmää lisättävän pisteen yhteydessä, pisteen tietoihin tallennetaan, että siitä on luotu särmä. Orientoidaan tahko $\mathbf{f}_1$ vastapäivään. Tallennetaan särmien $\mathbf{e}_4$ ja $\mathbf{e}_3$ tietoihin, että ne yhdistyvät luotuun tahkoon $\mathbf{f}_1$. Tallennetaan myös särmän $\mathbf{e}_0$ tietoihin, että se yhdistyy tahkoon $\mathbf{f}_1$ korvaamalla sen tiedoissa oleva näkyvä tahko $\mathbf{f}_0$ tahkolla $\mathbf{f}_1$. Näkyvä tahko korvataan sen takia, että $\mathbf{e}_0$ yhdistyy jo kahteen tahkoon ja näkyvä tahko poistetaan lopuksi.



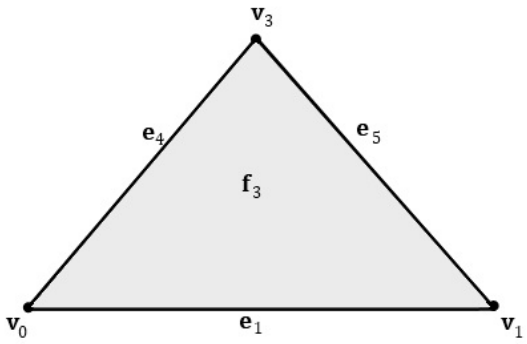
KUVA 20. Alkutilanne



KUVA 21. Tahkon $\mathbf{f}_1$ muodostaminen



KUVA 22. Tahkon $\mathbf{f}_2$ muodostaminen



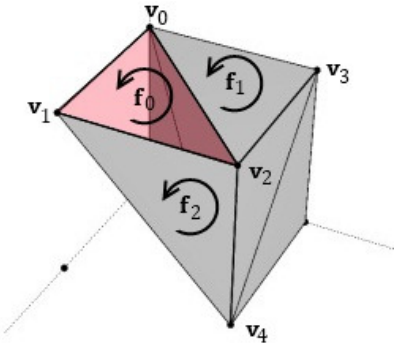
KUVA 23. Tahkon $\mathbf{f}_3$ muodostaminen

Seuraavaksi luodaan tahko $\mathbf{f}_2$ särmästä $\mathbf{e}_2$ ja pisteestä $\mathbf{v}_3$ (kuva 22). Nyt havaitaan, että särmän $\mathbf{e}_2$ päätepisteestä $\mathbf{v}_2$ on jo luotu särmä $\mathbf{e}_3$, joten asetetaan se kuulumaan myös tähän tahkoon. Näin saadaan yksi särmä kuulumaan kahteen tahkoon. Seuraavaksi poistetaan pisteen $\mathbf{v}_2$ tiedoista, että siitä on luotu särmä $\mathbf{e}_3$. Luodaan särmä $\mathbf{e}_5$ ja lisä-

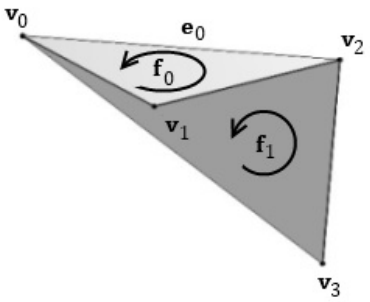
tään pisteen $\mathbf{v}_1$ tietoihin, että siitä on luotu särmä $\mathbf{e}_5$. Päivitetään särmien $\mathbf{e}_2$, $\mathbf{e}_3$ ja $\mathbf{e}_5$ tietoihin, että ne yhdistyvät tahkoon $\mathbf{f}_2$ ja orientoidaan $\mathbf{f}_2$ vastapäivään.

Seuraavaksi luodaan särmästä $\mathbf{e}_1$ ja pisteestä $\mathbf{v}_3$ tahko $\mathbf{f}_3$ (kuva 23). Nyt havaitaan, että särmän $\mathbf{e}_1$ molemmista päätepisteistä on luotu särmät lisättävään pisteeseen, joten kopioidaan ne tähän tahkoon ja poistetaan pisteiden tiedoista, että niistä on luotu särmät. Lopuksi päivitetään särmien $\mathbf{e}_1$, $\mathbf{e}_4$ ja $\mathbf{e}_5$ tietoihin, että ne yhdistyvät tahkoon $\mathbf{f}_3$ sekä orientoidaan $\mathbf{f}_3$ vastapäivään.

Pisteen tiedoista poistetaan särmä, jotta seuraavan pisteen lisäyksen yhteydessä ei voitaisi kopioida virheellisiä särmiä. Kun koko kierros on tehty eli kaikista särmistä jotka kuuluvat näkyvän alueen reunoihin on muodostettu uudet tahkot, ei minkään pisteen tietoihin jää tieto, että siitä on luotu särmä.



KUVA 24. Reuna ja orientaatio



KUVA 25. Uuden tahkon orientaatio

Uuden tahkon orientaatiossa on huomioitava, että särmästä ei voi päätellä, kuinka uusi kolmio on orientoitava. Kuva 24 havainnollistaa tätä tilannetta. Kuvassa 24 tahkossa $\mathbf{f}_1$ piste $\mathbf{v}_2$ on pisteen $\mathbf{v}_0$ jälkeen, kun taas luotavassa tahkossa $\mathbf{f}_0$ piste $\mathbf{v}_0$ on pisteen $\mathbf{v}_2$ jälkeen, joten joudutaan tutkimaan särmään yhdistyvää näkyvää tahkoa.

Tarkastellaan kuvan 25 tilannetta, jossa ollaan muodostamassa tahkoa $\mathbf{f}_0$ särmästä $\mathbf{e}_0$ ja pisteestä $\mathbf{v}_1$. Kuvassa 25 $\mathbf{f}_1$ on näkyvä tahko. Uusi tahko $\mathbf{f}_0$ täytyy orientoida niin, että lisättävästä pisteestä $\mathbf{v}_1$ katsottuna molemmat tahkot $\mathbf{f}_0$ ja $\mathbf{f}_1$ on orientoitu vastapäivään ja tämä saadaan seuraavalla tavalla: Valitaan särmän $\mathbf{e}_0$ ensimmäinen piste, joka voi olla $\mathbf{v}_0$ tai $\mathbf{v}_2$. Etsitään mikä piste tahkossa $\mathbf{f}_1$ on seuraavana särmän $\mathbf{e}_0$ ensimmäisestä päätepisteestä. Jos seuraava piste ei ole särmän $\mathbf{e}_0$ toinen päätepiste tiedetään, että tahkon $\mathbf{f}_0$ orientaatio on särmän $\mathbf{e}_0$ toinen päätepiste, ensimmäinen päätepiste ja lisättävä

piste $\mathbf{v}_1$. Muuten vaihdetaan kahden ensimmäisen paikkaa. Uuden tahkon luonti kuulostaa huomattavasti monimutkaisemmalta kuin mitä se käytännössä on. Pseudokoodi 2 esittää uuden tahkon luontia.

Pseudokoodi 2. Uuden tahkon luonti.

```
Syöte näkyvän alueen reunaan kuuluva särmä e ja lisättävä piste v
Paluu arvo, uusi tahko f

e0, e1 //Uuden tahkon särmät

for each särmän e päätepiste pi
    if pisteestä pi luotu särmä
        ei on tämä särmä.
        Poista tieto, että pisteestä pi on luotu särmä
    else
        ei on uusi särmä, jonka päätepisteet ovat v ja pi
        Lisää pisteen pi tietoihin, että siitä on luotu särmä ei
    end if
end for

Luo uusi tahko f, jonka särmät ovat e, e0 ja e1

// Orientoidaan tahko vastapäivään
fv = särmään e yhdistyvä näkyvä tahko
vn = Piste, joka on seuraavana tahkossa fv pisteestä p0

if vn on eri kuin p1
    Tahkon fv pisteiden järjestys on p1, p0 ja v
else
    Tahkon fv pisteiden järjestys on p0, p1 ja v
end if

 // Päivitetään särmien tietoihin, mihin tahkoihin ne yhdistyvät
Lisää särmien e0 ja e1 tietoihin, että ne yhdistyvät tahkoon f
Korvaa särmän e tiedoista tahko fv tahkolla f

Palauta f
```

5 KONVEKSISUUTEEN PERUSTUVAT MENETELMÄT

Tässä luvussa käsitellään kuinka törmäys voidaan havaita konvekseilla kappaleilla.

5.1 Erottava taso

Eräs tapa löytää erottava taso on tarkistaa, löytyykö kappaleesta A tahko, jonka ulkopuolella on kaikki kappaleen B pisteet. Jos tällainen tahko löytyi, niin kappaleet eivät

törmää, muuten ne törmäävät. Se, onko piste tahkon ulkopuolella vai ei, voidaan selvittää mm. kaavalla 11.

Pseudokoodi 3. Erottava taso.

```
for each f in F_A
    erottava taso = true

    for each v in V_B
        if Vol(v, f) > 0
            erottava taso = false
            break
        end if
    end for

    if erottava taso = true
        palauta ei törmäystä
    end if
end for

palauta törmäys
```

Kuten pseudokoodista huomaa, jos kappaleet eivät törmää, on iterointikierrosten määrä $|F_A| \cdot |V_B|$, missä $|F_A|$ on kappaleen A tahkojen lukumäärä ja $|V_B|$ on kappaleen B kärkipisteiden lukumäärä. Tämä on kohtalaisen yksinkertainen tapa selvittää törmäävätkö kappaleet, mutta kuten myöhemmin esitettävät algoritmit osoittavat, on nopeampia tapoja.

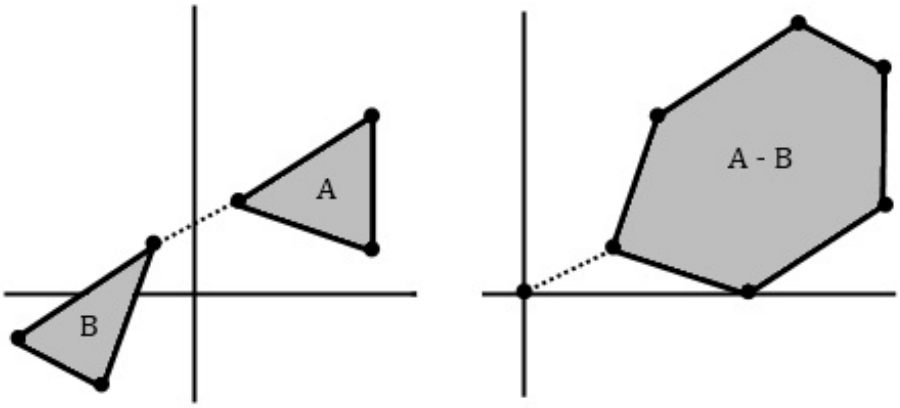
5.2 Minkowskin erotus

Minkowskin erotus on erittäin hyödyllinen ja käytetty menetelmä tunnistaessa kahden konveksin kappaleen törmäystä ja tähän perustuu myöhemmin esitettävät Gilbert-Johnson-Keerthi ja Chung-Wang algoritmit. Minkowskin erotus määritetään kaavalla 12.

$$A - B = \{\mathbf{a} - \mathbf{b} \mid \mathbf{a} \in A, \mathbf{b} \in B\} \tag{12}$$

Minkowskin erotuksella on ominaisuus, että kahden konveksin kappaleen Minkowskin erotus on myös konveksi kappale, ja tämän kappaleen etäisyys origosta on näiden kahden kappaleen välinen etäisyys. Jos origo on Minkowskin erotuksen sisällä, niin kappaleet ovat törmänneet. Tämä voidaan perustella sillä, että jos kaksi kappaletta

törmää, on niillä oltava vähintään yksi yhteinen piste ja tällöin näiden kahden pisteen erotus on nollavektori eli origo.



KUVA 26. Minkowskin erotus [1]

Alla on esitetty esimerkki Minkowskin erotuksesta. Vähennetään jokainen B:n piste jokaisesta A:n pisteestä.

Esimerkki 3. $\mathrm{A} = \{(1,2),(3,4)\}$

$\mathrm{B} = \{(1,0),(1,1)\}$

$\mathrm{A} - \mathrm{B} = \{(0,2),(2,4),(0,1),(2,3)\}$

5.3 Gilbert-Johnson-Keerthi-algoritmi

Tämä algoritmi perustuu Muratorin [4] esitykseen. Gilbert-Johnson-Keerthi-algoritmi(GJK) hyödyntää aiemmin mainittua Minkowskin erotusta. Ajatuksena on pyrkiä rakentamaan konveksin kappaleen sisälle kartio, joka sulkee origon sen sisälle. GJK-algoritmista on useita eri versioita ja tässä esitellään versio, joka palauttaa tiedon onko origo Minkowskin erotuksen sisällä vai ei. Määritellään aluksi funktio $S_C(\mathbf{d})$ seuraavasti:

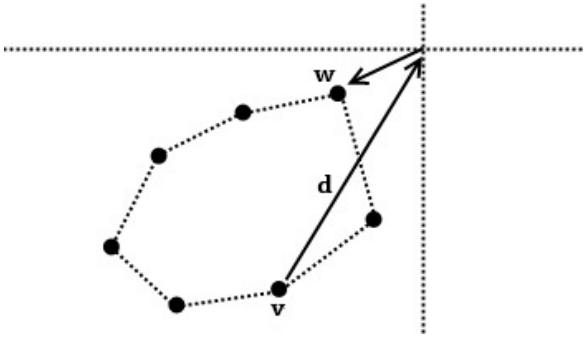
$$S_C(\mathbf{d}) = \mathbf{v} \text{ siten, että } \mathbf{v} \in \mathrm{C} \text{ ja } \mathbf{v} \bullet \mathbf{d} = \max\{\mathbf{v} \bullet \mathbf{d} : \mathbf{v} \in \mathrm{C}\} \qquad (13)$$

Funktio $S_C(\mathbf{d})$ palauttaa joukosta C sen pisteen, jonka pistetulo annetun vektorin **d** kanssa on suurin. Tämä vektori on myös etäisin vektori **d**:n osoittamasta suunnasta. GJK-algoritmin ei tarvitse laskea Minkowskin erotusta erikseen, koska Minkowskin

erotuksesta tarvitaan aina etäisin piste tietystä suunnasta, ja tämä voidaan laskea kaavalla 14.

$$S_{A-B}(\mathbf{d}) = S_A(\mathbf{d}) - S_B(-\mathbf{d}) \quad (14)$$

GJK-algoritmi etenee seuraavasti: Valitaan aluksi mielivaltainen piste Minkowskin erotuksesta, merkitään tätä **v**:llä. Lisätään tämä taulukkoon, jota kutsutaan simpleksiksi. Asetetaan etsintäsuunnaksi **d** = -**v**. Etsitään etäisin piste Minkowskin erotuksesta **d**:n suunnasta ja merkitään tätä **w**:llä. Jos **w** • **d** < 0, niin origo ei ole Minkowskin erotuksen sisällä, joten palautetaan, *ei törmäystä*. Tätä voidaan havainnollistaa sillä, että valitsemalla mikä tahansa origosta poikkeava piste Minkowskin erotuksesta ja etsimällä etäisin piste origon suunnasta. Tästä pisteestä katsottuna täytyy päästä origon ohi, jotta origo olisi Minkowskin erotuksen sisällä.



KUVA 27. Origo Minkowskin erotuksen ulkopuolella

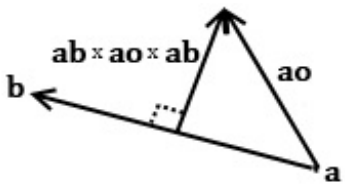
Jos **w** • **d** ≥ 0, niin lisätään **w** simpleksiin. Seuraavaksi kutsutaan alialgoritmia, joka päivittää etsintäsuunnan, mahdollisesti simpleksin ja tiedon onko origo kartion sisällä. Alialgoritmi käsittelee kolmea eri tilannetta seuraavasti: simpleksi on jana, kolmio tai kartio. Sen jälkeen kun alialgoritmia on kutsuttu, etsitään etäisin piste sen palauttamasta etsintäsuunnasta ja toistetaan edelliset askeleet kunnes **w** • **d** < 0. Tällöin törmäystä ei tapahtunut tai alialgoritmi ilmoittaa, että origo on kartion sisällä. Tällöin törmäys on tapahtunut. Pääalgoritmin tarkoitus on vain lisätä uusi piste simpleksiin ja palauttaa tieto törmäyksestä. Pseudokoodi 4 esittää GJK-algoritmin pääalgorimia.

Pseudokoodi 4. GJK-algoritmi.

```
Simpleksi on tyhjä taulukko
v = Mikä tahansa piste Minkowskin erotuksesta
if v on origo
    Palauta törmäys
end if

d = -v
Lisää v simpleksiin

while origo ei ole kartion sisällä
    w = S_A-B(d)
    if w on origo
        Lopeta palauta törmäys
    end if
    if w • d < 0
        Lopeta ja palauta ei törmäystä
    end if
    Lisää w simpleksiin
    Kutsu alialgoritmia
end while

Palauta törmäys
```

Origon etsintä janan tapauksessa

Janan tapauksessa päivitetään ainoastaan etsintäsuunta, josta origoa lähdetään etsimään. Kuvassa 28 piste **b** on ensimmäisenä lisätty piste. Tämän algoritmin yhteydessä **a**:lla merkitään aina viimeksi lisättyä pistettä. Pisteiden järjestyksellä on merkitystä kolmion ja kartion tapauksessa. Janan tapauksessa origoa lähdetään etsimään suunnasta **ab** × **ao** × **ab**.

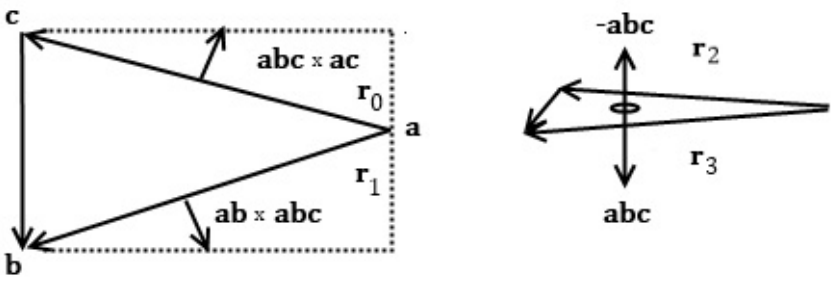


KUVA 28. Janasta seuraava etsintäsuunta [4]

Origon etsintä kolmion tapauksessa

Kolmion käsittely on kolmesta vaiheesta monimutkaisin. Kolmiosta etsitään, onko sen sisäosa lähinnä origoa vai joku sen sivuista. Tarkastellaan kuvan 29 tilannetta. Tiedetään että, origo on sivun **cb** oikealla puolella, koska muuten pistettä **a** ei olisi lisätty simpleksiin. Tiedetään, että origo on vasemmalla puolella pistettä **a**, koska etsittiin

etäisin piste origon suunnasta sivusta **cb** katsottuna. Origon täytyy myös olla sivun **cb** välissä, joten origo on jossain katkoviivojen esittämän alueen sisällä.



KUVA 29. Kolmion käsittely [4]

Nyt täytyy päätellä, mikä osa kolmiosta on lähinnä origoa. Osat ovat sivut **cb**, **ac**, ja **ab** sekä kolmion sisäosa, jota kaikki kolme pistettä esittävät. Näistä voidaan heti sulkea pois sivun **cb**, koska tiedetään, että origo on sen oikealla puolella, joten sivu **cb** ei voi olla sitä lähimpänä.

Helpoin tapa lähteä etsimään, mikä kolmion osista on lähinnä origoa, on aloittaa sivulla. Aloitetaan sivulla **ac**. Jos (**abc** × **ac**) • **ao** > 0, niin sivu **ac** on lähinnä origoa, joten origo on kuvan 28 tapauksessa alueella $\mathbf{r}_0$. Asetetaan simpleksi sivuksi **ac** ja etsintäsuunta vektoriksi **ac** × **ao** × **ac**. Simpleksin pisteiden järjestykseksi asetetaan **c, a**. Simpleksin pisteiden järjestys on tärkeä, jotta seuraavan kerran kolmion yhteydessä voidaan päätellä lähin osa ja etsintäsuunta. Toinen tärkeä huomio on se, että vektoria **abc** × **ac** ei voi asettaa etsintäsuunnaksi, koska se on kolmion määräämässä tasossa ja origo voi olla eri tasossa. Tällöin etsintäsuunta olisi väärä. Vektorilla **abc** × **ac** tarkistettiin vain onko sivu **ac** lähinnä origoa.

Jos sivu **ac** ei ole lähinnä origoa, tarkistetaan sivu **ab**. Jos (**ab** × **abc**) • **ao** > 0, niin sivu **ab** on lähinnä origoa. Asetetaan etsintäsuunnaksi vektori **ab** × **ao** × **ab** ja simpleksi sivuksi **ab**, jossa pisteiden järjestys on **b**, **a**.

Jos kumpikaan sivuista ei ole lähinnä origoa, on kolmion sisäosa lähinnä. Nyt täytyy päätellä, kummalla puolella se on. Jos **abc** • **ao** > 0 origo on kuvan 29 tapauksessa alueella $\mathbf{r}_3$, joten asetetaan etsintäsuunnaksi vektori **abc**. Simpleksistä muutetaan ainoastaan pisteiden järjestystä. Järjestykseksi asetetaan **b**, **c**, **a**. Järjestys muutetaan, koska seuraava piste, joka lisätään simpleksiin tekee siitä kartion. Näin saadaan jokainen

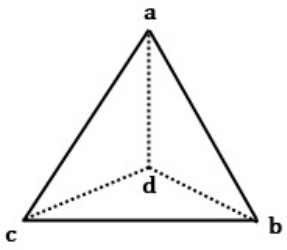
kolmio kierrettyä vastapäivään. Jos origo on kuvan 29 kolmion yläpuolella alueella $\mathbf{r}_2$, asetetaan etsintäsuunnaksi vektori **–abc**. Simpleksille ei tehdä mitään. Pseudokoodissa 5 on esitetty kolmion käsittely.

Pseudokoodi 5. GJK-algoritmi, kolmion käsittely.

```
ab = b - a
ac = c - a
abc = ab × ac

if (abc × ac) • ao > 0
    Simpleksi on [c, a]
    Etsintäsuunta on ac × ao × ac
end if

else if (ab × abc) • ao > 0
    Simpleksi on [b, a]
    Etsintäsuunta on ab × ao × ab
end if

else if abc • ao > 0
    Simpleksi on [b, c, a]
    Etsintäsuunta on abc
end if

else
    Etsintäsuunta on -abc
end if
```

Origon etsintä kartion tapauksessa

Kartion tapauksessa palautetaan tieto törmäyksestä mikäli origo on kartion sisällä. Jos origo ei ole kartion sisällä, tarkistetaan mikä kartion kolmioista on lähinnä origoa ja asetetaan simpleksiksi tämä kolmio ja siirrytään käsittelemään kolmiota.



KUVA 30. Kartio

Se, kuinka päätellään onko origo kartion sisällä tai mikä kolmioista on lähinnä, voidaan tehdä seuraavasti: Jokaiselle kolmiolle lasketaan kolmionpinnan normaali **n**, jos

ao • **n** > 0, jollekin kolmiolle on origo kartion ulkopuolella ja tämä kolmio on lähinnä origoa. Simpleksi asetetaan täksi kolmioksi ja siirrytään kolmion käsittelyyn. Jos ei yhdenkään kolmionpinnan normaalin pistetulo vektorin **ao** kanssa ole suurempi kuin nolla, on origo kartion sisällä. Kuvassa 30 on esitetty tämä tilanne. Tästä on hyvä huomata, että kolmiota (**c**, **d**, **b**) ei tarvitse tarkistaa, koska tiedetään, että origo on sen yläpuolella. Pseudokoodissa 6 on esitetty kartion käsittely.

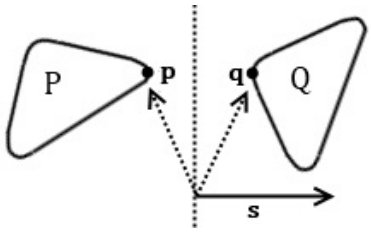
Pseudokoodi 6. GJK-algoritmi, kartion käsittely.

```
ao = -a

bac = (a – b) × (c – b)
if bac • ao > 0
    Simpleksi on (c, b, a)
    Siirry kolmion käsittelyyn
end if

cad = (a – c) × (d – c)
if cad • ao > 0
    Simpleksi on (d, c, a)
    Siirry kolmion käsittelyyn
end if

dab = (a – d) × (b – d)
if dab • ao > 0
    Simpleksi on (b, d, a)
    Siirry kolmion käsittelyyn
end if

Palauta törmäys
```

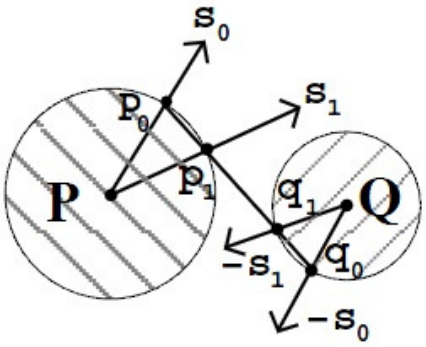
5.4 Chung-Wang-algoritmi

Tämä luku perustuu lähteisiin [5] ja [6]. Chung-Wang-algoritmi koostuu kahdesta algoritmista. Pääalgoritmi havaitsee, ovatko kappaleet erillään ja alialgoritmi tutkii, törmäävätkö kappaleet.

Pääalgoritmi pyrkii etsimään vektorin **s**, jonka suunnassa kappaleet ovat erillään. Kuva 31 havainnollistaa tätä. Jos kappaleet eivät ole erillään suunnassa **s**, lasketaan uusi mahdollinen erottava vektori $\mathbf{s}_{i+1}$, ja tarkistetaan ovatko kappaleet erillään siinä suunnassa. Tätä toistetaan kunnes on löydetty erottava vektori tai alialgoritmi palauttaa törmäyksen.



KUVA 31. Erottava vektori

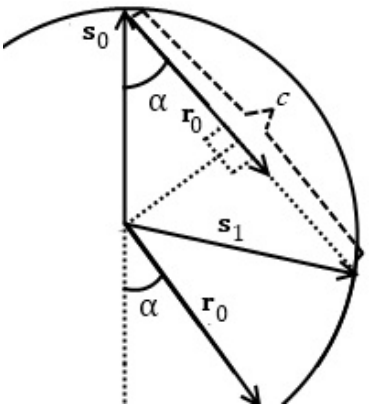


KUVA 32. Kaksi palloa [5]

Pääalgoritmi etenee seuraavasti: Valitaan aluksi mielivaltainen vektori $\mathbf{s}_i$. Etsitään kappaleista P ja Q pisteet $\mathbf{p}_i$ ja $\mathbf{q}_i$ niin, että $\mathbf{p}_i = S_P(\mathbf{s}_i)$ ja $\mathbf{q}_i = S_Q(-\mathbf{s}_i)$. Nämä pisteet ovat kappaleiden ääripisteet vastakkaisissa suunnissa. Jos $\mathbf{s} \bullet \mathbf{p}_i \leq \mathbf{s} \bullet \mathbf{q}_i$, niin kappaleet eivät törmää joten palautetaan, *ei törmäystä*. Muuten lasketaan uusi mahdollinen erottava vektori kaavalla 15 ja etsitään etäisimmät pisteet $\mathbf{p}_{i+1}$ ja $\mathbf{q}_{i+1}$. Lisäksi tarkistetaan onko $\mathbf{s}_{i+1} \bullet \mathbf{p}_{i+1} \leq \mathbf{s}_{i+1} \bullet \mathbf{q}_{i+1}$. Tätä toistetaan niin kauan, kunnes löydetään erottava vektori, tai alialgoritmi palauttaa törmäyksen. Tämän algoritmin yhteydessä kappaleet eivät törmää, jos kappaleiden välinen etäisyys on nolla.

$$\mathbf{s}_{i+1} = \mathbf{s}_i - 2(\mathbf{r}_i \bullet \mathbf{s}_i)\mathbf{r}_i \text{ , missä } \mathbf{r}_i = (\mathbf{q}_i - \mathbf{p}_i)/||\mathbf{q}_i - \mathbf{p}_i|| \quad (15)$$

Kaavan 15 idea perustuu siihen, että jos kaksi palloa ei törmää ja vektori $\mathbf{s}_0$ ei ole erottava, niin vektori $\mathbf{s}_1$ on erottava vektori. Jos kahden erillään olevan pallon pisteiden $\mathbf{p}_0$ ja $\mathbf{q}_0$ välille vedetään viivaa, niin pallon geometristen ominaisuuksien vuoksi erottava vektori osoittaa kohtaan, jossa viiva leikkaa pallon P (kuva 32). Kaikki pisteet $\mathbf{p}_0$, $\mathbf{q}_0$, $\mathbf{p}_1$ ja $\mathbf{q}_1$ ovat samalla viivalla.



KUVA 33. $\mathbf{s}_1$ on erottava vektori

Kaava 15 voidaan johtaa seuraavalla tavalla (kuva 33): Tiedetään, että $\mathbf{s}_1 = \mathbf{s}_0 + c\mathbf{r}_0$, jossa c on tuntematon vakio. c on vektorien $\mathbf{s}_0$ ja $\mathbf{s}_1$ välinen etäisyys ja se saadaan kaavalla 16.

$$\frac{c}{2} = \cos(\alpha)||\mathbf{s}_0|| \tag{16}$$

Koska vektorit $\mathbf{s}_0$, $\mathbf{s}_1$ ja $\mathbf{r}_0$ ovat yksikkövektoreita, voidaan yhtälö kirjoittaa seuraavalla tavalla:

$$c = 2(-\mathbf{s}_0 \bullet \mathbf{r}_0) = -2(\mathbf{s}_0 \bullet \mathbf{r}_0)\text{, missä } \mathbf{r}_0 = (\mathbf{q}_0 - \mathbf{p}_0)/||\mathbf{q}_0 - \mathbf{p}_0|| \tag{17}$$

Pseudokoodi 7 esittää Chung-Wang-algoritmin toiminnan. Algoritmin alussa haetaan edellinen erottava vektori **s** muistista, koska edellinen erottava vektori on useimmiten hyvin lähellä uutta mahdollista erottavaa vektoria. Alialgoritmin idea on tarkistaa, onko olemassa sellaista vektoria **w** niin, että $\mathbf{w} \bullet \mathbf{r}_i \geq 0$, jokaiselle $\mathbf{r}_i$:lle. Jos tällaista vektoria ei löydy, origo on Minkowskin erotuksen sisällä, koska vektorit $\mathbf{r}_i$ jakautuvat yli yhden pallonpuoliskon, joten palautetaan *törmäys*. Pseudokoodi sisältää myös kaksi lisäehtoa. Ensimmäinen on lisälopetusehto $\mathbf{r}_{i-1} = \mathbf{r}_i$. Jos vektori $\mathbf{r}_i$ toistuu peräkkäin, palautetaan, *ei törmäystä*. Toinen on etsintäsuunnan vaihto, jos tietyn iterointikierroksen jälkeen $\mathbf{r}_i$ toistuu, mutta ei peräkkäin. Alialgoritmi ja lopetusehdot tarkastellaan tarkemmin myöhemmin.

Pseudokoodi 7. Chung-Wang-pääalgoritmi.

```
Hae s muistista
R on tyhjä taulukko

while
    p_i = S_P(s)
    q_i = S_Q(-s)

    if p • s ≤ q • s
        Palauta ei törmäystä
    else if r_i-1 = r_i
        Palauta ei törmäystä
    else if tietty määrä kierroksia tehty ja r_i toistuu
        s = w
    else
        if Pisteet jakautuvat yli yhden pallon puoliskon
            Palauta törmäys
        else
            r_i = (q_i – p_i) / ||( q_i – p_i)||
            s = s – 2(r_i • s) r_i
            Lisää r_i taulukkoon R
        end if
    end if
end while
```

5.4.1 Alialgoritmi

Alialgoritmin tarkoitus on tarkistaa onko origo Minkowskin erotuksen sisällä. Jokainen vektori $\mathbf{r}_i$ on yksi Minkowskin erotuksen vektoreista yksikkövektorina. Se, onko origo Minkowskin erotuksen sisällä, tehdään etsimällä vektori **w** jolle
$\forall\ \mathbf{r} \in R$: $\mathbf{w} \bullet \mathbf{r} \geq 0$. Jos tällainen vektori on olemassa, niin origo ei ole Minkowskin erotuksen sisällä.

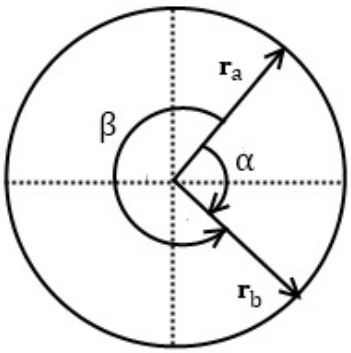
Jos origo on Minkowskin erotuksen sisällä, niin pisteiden täytyy jakautua yli yhden pallonpuoliskon. Ongelma on tarkistaa jakautuvatko pisteet yli yhden pallonpuoliskon. Jos pisteet ovat samalla pallon puoliskolla, niin vektori **w** on olemassa. Ongelma voidaan siirtää kaksiulotteiseksi projisoimalla pisteet tasoon. Jos on olemassa viiva, joka kulkee origon kautta ja kaikki pisteet ovat sen yhdellä puolella, ts. kaikki pisteet ovat samalla ympyrän puoliskolla, niin kaikki pisteet ovat myös samalla pallonpuoliskolla.

Projisointi tehdään etsimällä matriisi **M**, joka muuntaa vektorin $\mathbf{r}_i$ z-akseliksi (0, 0, 1). Tämän jälkeen vektorit $\mathbf{r}_2 \ldots \mathbf{r}_{i-1}$ kerrotaan matriisilla **M**, jonka jälkeen projektio saa-

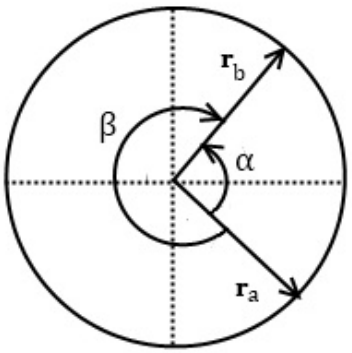
daan nollaamalla z-arvot vektoreista. Vektoria $\mathbf{r}_i$ ei tarvitse projisoida, koska se projisoituu origoksi. Kaava 18 esittää muunnosmatriisia.

$$\begin{bmatrix} xz/d & -y/d & x \\ yz/d & x/d & y \\ -(x^2+y^2)/d & 0 & z \end{bmatrix} \quad d = \sqrt{x^2+y^2+z^2} \tag{18}$$

Erottavan viivan etsimiseen käytetään seuraavaa tietoa hyväksi. Olkoon kaksi vektoria $\mathbf{r}_a$ ja $\mathbf{r}_b$ yksikköympyrässä. Jos $x_{ra} \cdot y_{rb} - x_{rb} \cdot y_{ra} < 0$, niin kulma vektorista $\mathbf{r}_a$ kierrettynä myötäpäivään vektoriin $\mathbf{r}_b$ on pienempi kuin kulma vektorista $\mathbf{r}_a$ vastapäivään vektoriin $\mathbf{r}_b$ (kuva 34). Jos $x_{ra} \cdot y_{rb} - x_{rb} \cdot y_{ra} > 0$, niin kulma vektorista $\mathbf{r}_a$ vastapäivään vektoriin $\mathbf{r}_b$ on pienempi kuin kulma vektorista $\mathbf{r}_a$ myötäpäivään vektoriin $\mathbf{r}_b$ (kuva 35). Kulmat ovat itseisarvoja.



KUVA 34. $x_{ra} \cdot y_{rb} - x_{rb} \cdot y_{ra} < 0$



KUVA 35. $x_{ra} \cdot y_{rb} - x_{rb} \cdot y_{ra} > 0$

Tällä tiedolla voidaan ongelma ratkaista seuraavalla tavalla: Valitaan vektorit $\mathbf{r}_a$ ja $\mathbf{r}_b$ niin, että $\mathbf{r}_a = \mathbf{M} \cdot \mathbf{r}_0$ ja $\mathbf{r}_b = \mathbf{M} \cdot \mathbf{r}_1$. Orientoidaan vektorit $\mathbf{r}_a$ ja $\mathbf{r}_b$ niin, että $\mathbf{r}_b$ on myötäpäivään vektorista $\mathbf{r}_a$, kuten kuvassa 34. Tämä voidaan tehdä tarkistamalla onko $x_{ra} \cdot y_{rb} - x_{rb} \cdot y_{ra} > 0$. Tällöin ollaan kuvan 35 tilanteessa, joten vaihdetaan vektorien paikkaa. Muuten ei tehdä mitään. Seuraavaksi jokaiselle $\mathbf{r}_2 \ldots \mathbf{r}_{i-1}$ tehdään seuraava:

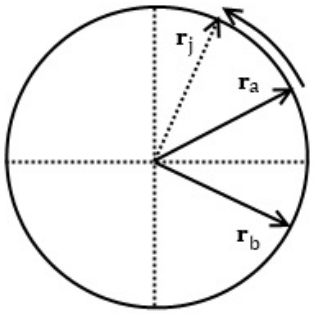
1. Siirretään vektori $\mathbf{r}_i$ samaan koordinaatistoon, missä $\mathbf{r}_a$ ja $\mathbf{r}_b$ ovat, laskemalla $\mathbf{r}_j = \mathbf{M} \cdot \mathbf{r}_i$. Nyt projektio tasoon saadaan pudottamalla vektorin $\mathbf{r}_j$ z-komponentti pois.

2. Jos $x_{ra} \cdot y_{rj} - y_{ra} \cdot x_{rj} < 0$ ja $x_{rb} \cdot y_{rj} - y_{rb} \cdot x_{rj} > 0$, niin vektori $\mathbf{r}_j$ on vektorien $\mathbf{r}_a$ ja $\mathbf{r}_b$ välissä, joten ei tehdä mitään. Siirrytään kohtaan yksi käsittelemään seuraavaa vektoria $\mathbf{r}_{j+1}$.

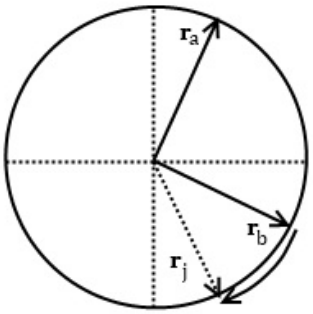
3. Jos $x_{ra} \cdot y_{rj} - y_{ra} \cdot x_{rj} > 0$ ja $x_{rb} \cdot y_{rj} - y_{rb} \cdot x_{rj} > 0$, asetetaan $\mathbf{r}_a = \mathbf{r}_j$ ja siirrytään kohtaan yksi käsittelemään seuraavaa $\mathbf{r}_{j+1}$ vektoria, kuva 36.

4. Jos $x_{ra} \cdot y_{rj} - y_{ra} \cdot x_{rj} < 0$ ja $x_{rb} \cdot y_{rj} - y_{rb} \cdot x_{rj} < 0$, asetetaan $\mathbf{r}_b = \mathbf{r}_j$ ja siirrytään kohtaan yksi käsittelemään seuraavaa $\mathbf{r}_{j+1}$ vektoria, kuva 37.

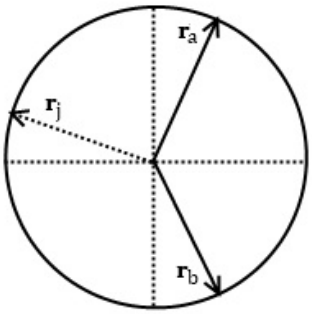
5. Jos tänne asti on tultu, pienin kulma $\mathbf{r}_a$:n ja $\mathbf{r}_j$:n välillä on $\mathbf{r}_a$:sta vastapäivään ja pienin kulma $\mathbf{r}_b$:n ja $\mathbf{r}_j$:n välillä on $\mathbf{r}_b$:sta myötäpäivään. Tällöin pisteet eivät jakaudu yhdelle ympyrän puoliskolle, joten palautetaan, että vektoria **w** ei ole olemassa, kuva 38.



KUVA 36. Tilanne 3



KUVA 37. Tilanne 4



KUVA 38. Tilanne 5

Jos kohtaan viisi ei menty minkään $\mathbf{r}_j$:n kohdalla, niin kaikki $\mathbf{r}_j$ vektorit ovat samalla ympyrän puoliskolla, koska erottava viiva on olemassa. Joten palautetaan, että vektori **w** on olemassa. Alialgoritmissa tallennetaan joka kerta vektori **w**, joka on $\mathbf{M}^{-1} \cdot (\mathbf{r}_a + \mathbf{r}_b)/\|\mathbf{r}_a + \mathbf{r}_b\|$, jotta ei jokaisen lisätyn pisteen kohdalla tarvitsisi suorittaa aiemmin mainittua menetelmää. Tallentamalla **w**, voidaan jokaisen lisätyn pisteen kohdalla ensin tarkistaa onko $\mathbf{w} \bullet \mathbf{r}_i > 0$. Tällöin vektori **w** on olemassa. Pseudokoodi 8 esittää alialgoritmia.

Pseudokoodi 8. Chung-Wang-alialgoritmi.

```
Syöte pistejoukko R
Paluu arvo r ϵ R: w • r ≥ 0

if |R| = 3
    w = (r0 + r1) / ||(r0 + r1)||
end if

if w • ri >= 0
    Palauta tosi
end if

Laske matriisi M niin, että M · ri = ẑ
ra = M · r0
rb = M · r1

if ra.x · rb.y – rb.x · ra.y > 0
    vaihda(ra, rb)
end if

for each r2 … ri-1 in R
    rj = M · ri
    rarj = ra.x · rj.y - ra.y · rj.x
    rbrj = rb.x · rj.y - rb.y · rj.x

    if rarj < 0 and rbrj > 0
        continue
    else if rarj > 0 and rbrj > 0
        ra = rj
    else if rarj < 0 and rbrj < 0
        rb = rj
    else
        Palauta epätosi
    end if
end for

w = M^-1 · (ra + ra) / ||ra + ra||
Palauta tosi
```

5.4.2 Lisäehdot lopettamiselle

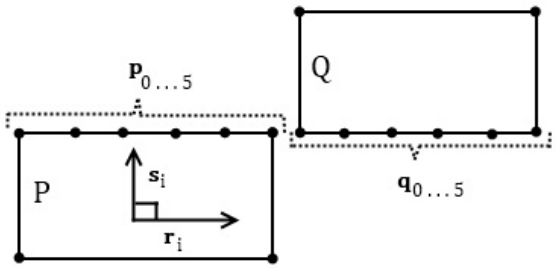
Algoritmilla on ominaisuus, että jos **p** ja **q** toistuvat peräkkäin, niin silloin $\mathbf{r}_i = \mathbf{r}_{i+1}$, $\mathbf{p}_i = \mathbf{p}_{i+1}$ ja $\mathbf{q}_i = \mathbf{q}_{i+1}$. Tällöin kappaleet eivät törmää, mikä voidaan osoittaa seuraavasti:

$$\begin{aligned}\mathbf{s}_{i+1} \bullet \mathbf{r}_{i+1} &= \mathbf{r}_{i+1} \bullet (\mathbf{s}_i - 2(\mathbf{r}_i \bullet \mathbf{s}_i)\mathbf{r}_i) \\ &= \mathbf{s}_i \bullet \mathbf{r}_{i+1} - 2(\mathbf{r}_i \bullet \mathbf{s}_i)(\mathbf{r}_i \bullet \mathbf{r}_{i+1}) \\ &= \mathbf{s}_i \bullet \mathbf{r}_{i+1} - 2(\mathbf{r}_i \bullet \mathbf{s}_i)(\mathbf{r}_i \bullet \mathbf{r}_i) \\ &= \mathbf{s}_i \bullet \mathbf{r}_i - 2(\mathbf{r}_i \bullet \mathbf{s}_i)\end{aligned} \tag{19}$$

$$= -\mathbf{s}_i \bullet \mathbf{r}_i$$

Tästä voidaan päätellä, että kappaleet eivät törmää, koska edellisellä kierroksella $\mathbf{s}_i \bullet \mathbf{r}_i$ oli pienempi kuin nolla ja nyt $\mathbf{s}_{i+1} \bullet \mathbf{r}_{i+1}$ on vastakkaismerkkinen.

Toinen lisäehto on etsintäsuunnan vaihto, jotta algoritmi ei jäisi ikuiseen silmukkaan. Jos $\mathbf{r}_i$ toistuu tietyn iterointikierroksen jälkeen, mutta ei peräkkäin, asetetaan etsintäsuunnaksi $\mathbf{s}_{i+1} = \mathbf{w}$. Tämän tilanteen havainnollistaa kuva 39, missä $\mathbf{p}_i$ suunnasta $\mathbf{s}_i$ voi olla mikä tahansa pisteistä $\mathbf{p}_{0\ldots5}$, ja samoin $\mathbf{q}_i$ voi olla mikä tahansa pisteistä $\mathbf{q}_{0\ldots5}$ suunnasta - $\mathbf{s}_i$. Ideaalitilanteessa $\mathbf{s}_i \bullet \mathbf{r}_i = 0$, jolloin algoritmi päättyy, mutta liukulukuepätarkkuuksista johtuen ei voida luottaa siihen. Jos $\mathbf{s}_i \bullet \mathbf{r}_i > 0$, mutta hyvin pieni luku, niin $\mathbf{s}_{i+1} \approx \mathbf{s}_i$. Joten seuraavat $\mathbf{p}_{i+1}$ ja $\mathbf{q}_{i+1}$ ovat jälleen mitkä tahansa pisteistä $\mathbf{p}_{0\ldots5}$ ja $\mathbf{q}_{0\ldots5}$, mutta samat eivät välttämä toistu, joten lisälopetusehto $\mathbf{r}_i = \mathbf{r}_{i+1}$ ei sitä havaitse. Pahimmassa mahdollisessa tilanteessa joudutaan ikuiseen silmukkaan. Asettamalla seuraavaksi etsintäsuunnaksi $\mathbf{s}_{i+1} = \mathbf{w}$, vältytään ikuiselta silmukalta.[5][6].



KUVA 39. Etsintäsuunnan vaihto

5.4.3 Chung-Wang-algoritmin ongelmat

Van Den Bergen osoitti [7], että todistus Chung-Wang-algoritmin lähenemisestä kohti erottavaa vektoria on virheellinen. Todistus, johon hän viittasi [6], etenee seuraavasti: Olkoon $\mathbf{w}$ kappaleiden P ja Q erottavavektori, jolloin $\mathbf{s}_i$ ja $\mathbf{s}_{i+1}$ eivät ole erottavia vektoreita. Tästä seuraa, että

$$\mathbf{s}_{i+1} \bullet \mathbf{w} > \mathbf{s}_i \bullet \mathbf{w} \tag{20}$$

Joka voidaan johtaa seuraavalla tavalla:

$$\mathbf{s}_{i+1} \bullet \mathbf{w} = \mathbf{s}_i \bullet \mathbf{w} - 2(\mathbf{r}_i \bullet \mathbf{s}_i)(\mathbf{r}_i \bullet \mathbf{w}) \quad (21)$$

Koska $\mathbf{r}_i \bullet \mathbf{w} > 0$ ja $\mathbf{r}_i \bullet \mathbf{s}_i < 0$ tästä seuraa, että

$$\mathbf{s}_{i+1} \bullet \mathbf{w} - \mathbf{s}_i \bullet \mathbf{w} = -2(\mathbf{r}_i \bullet \mathbf{s}_i)(\mathbf{r}_i \bullet \mathbf{w}) > 0 \quad (22)$$

Van Bergenin mukaan $\mathbf{r}_i \bullet \mathbf{w}$ voi olla nolla ja tämä johtuu siitä, että $\mathbf{s}_i$ ja $\mathbf{s}_{i+1}$ ovat yksikkövektoreita. Jos $\mathbf{r}_i \bullet \mathbf{w} = 0$, tästä seuraa, että $\mathbf{s}_{i+1} = \mathbf{s}_i$. Joten ei olla yhtään lähempänä erottavaa vektoria. Van Bergen kuitenkin totesi, että etsintäsuunnan vaihto korjaa tämän ongelman. Kokeellisesti on havaittu, että tietyn muotoiset kappaleet voivat aiheuttaa lähes mielivaltaisen määrään iterointeja ennen kuin erottava vektori on löytynyt. On myös havaittu, että kun kappaleet ovat todella lähellä toisiaan, algoritmi toimii erittäin hitaasti. Nämä ongelmat ovat kuitenkin ääritapauksia ja käytännössä algoritmi on havaittu nopeaksi.[7.]

6 DYNAAMINEN TÖRMÄYKSEN TUNNISTUS

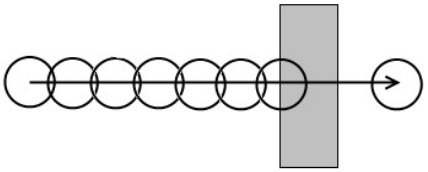
Tähän mennessä käsitellyt menetelmät havaitsevat vain törmäävätkö kaksi paikallaan olevaa kappaletta. Dynaamisessa törmäyksen tunnistuksessa otetaan mukaan liike.

Useimmiten käytännössä pelkällä staattisella törmäyksen tunnistuksella pärjää, varsinkin peliohjelmoinnissa, jossa pelisilmukka suoritetaan kymmeniä kertoja sekunnissa. Tällöin kappaleet eivät liiku kovin suuria etäisyyksiä kahden pelisilmukan suoritusten välillä. Esim. jos pikajuoksija juoksee nopeudella 10 m/s ja pelisilmukka suoritetaan 60 kertaa sekunnissa, liikkuu pikajuoksija yhden silmukan suorituksenaikana noin 16,7 cm, mikä on tässä tilanteessa törmäyksen tunnistuksen virhemarginaalin yläraja. Peliohjelmoinnin kannalta 16,7 cm ihmiseen suhteutettuna on varsin siedettävä virhemarginaali, eikä monissa suhteellisen moderneissa peleissä ole tätä tarkempaa törmäyksen tunnistusta. Staattista törmäyksen tunnistusta voidaan parantaa nostamalla pelisilmukan suoritusten määrää, mutta tämä puolestaan tekee pelistä raskaamman.

Suurten nopeuksien yhteydessä on otettava liike huomioon jollain muulla tavalla. Helpoin tapa ottaa liike huomioon on suorittaa joukko staattisia tömäyksentunnistuksia kappaleen alku- ja loppusijainnin väliltä, ts. näytteistää. Tässä tulee vastaan ongelma nimeltä tunnelointi. Tunneloinnilla tarkoitetaan, että jos suoritetaan liian vähän

staattisia törmäyksen tunnistuksia, niin hypätään esteen yli ja törmäystä ei havaita, kuvan 40 osoittamalla tavalla. Toinen ongelma on se, että jos suoritetaan liian monta törmäyksen tunnistusta, kärsii suorituskyky. [1, s. 214 – 215.]

KUVA 40. Liian vähän näytteitä [1]

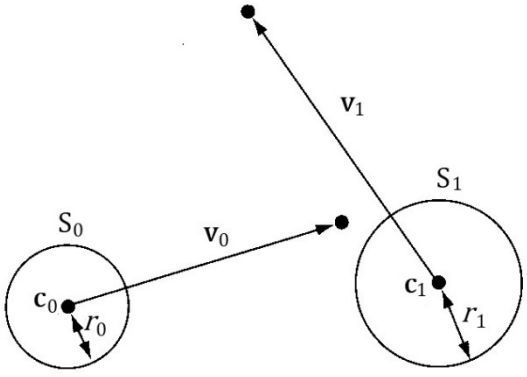


KUVA 41. Liian paljon näytteitä [1]

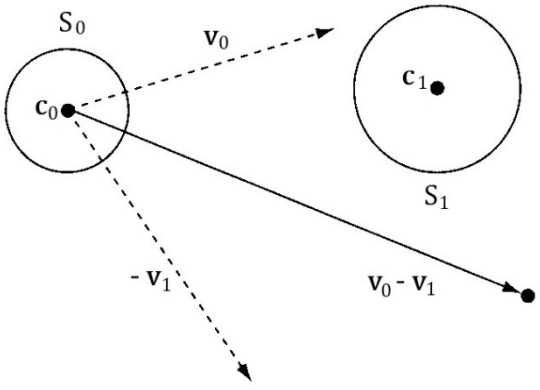
Seuraavaksi käsitteellään kuinka dynaaminen törmäyksen tunnistus voidaan tehdä ilman staattisia törmäyksen tunnistuksia. Ensin tutkitaan kuinka havaitaan törmäävätkö kaksi liikkuvaa palloa ja sitten esitellään tapa, jolla liike yhdistetään GJK-algoritmiin. Molemmissa menetelmissä oletetaan, että liike on suoraviivaista ja lineaarista, kuten se peliohjelmoinnissa yleensä on.

6.1 Kaksi liikkuvaa palloa

Tämä luku perustuu Ericsonin kirjaan [1]. Kahden liikkuvan kappaleen yhteydessä on hyvä tietää, että ongelma saadaan aina muotoon, jossa toinen on paikallaan oleva kappale ja toinen liikkeessä oleva kappale. Tarkastellaan kuvan 42 tilannetta jossa on kaksi liikkuvaa palloa S_0 ja S_1 sekä niitä vastaavat liikevektorit $\mathbf{v}_0$ ja $\mathbf{v}_1$. Asettamalla pallo S_1 paikallaan olevaksi ja muuttamalla pallon S_0 liikkeen suunnaksi $\mathbf{v}_0 - \mathbf{v}_1$, ongelma saadaan tilanteeseen, jossa toinen pallo on paikallaan oleva ja toinen liikkeessä.

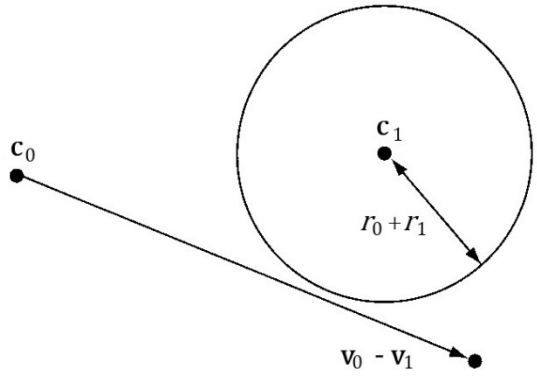


KUVA 42. Alkutilanne



KUVA 43. Liikevektorien erotus

Kahden pallon tapauksessa ongelma helpottuu entisestään. Ongelma saadaan muotoon mikä on janan, jonka alkupiste on $\mathbf{c}_0$ ja päätepiste on $\mathbf{c}_0 + (\mathbf{v}_0 - \mathbf{v}_1)$ etäisyys pallosta. Pallon keskipiste on $\mathbf{c}_1$ ja säde on $r_0 + r_1$. Jos janan etäisyys pallosta on pienempi kuin $r_0 + r_1$, niin pallot törmäävät. Tämä voidaan laskea siten, että lasketaan ensin mikä on lähin piste $\mathbf{p}$ janasta pisteeseen $\mathbf{c}_1$ kohdassa 3 esitellyllä tavalla. Jos $\|\mathbf{p} - \mathbf{c}_1\| \leq r_0 + r_1$, niin pallot törmäävät (kuva 44).



KUVA 44. Jana ja pallo

Toinen tapa laskea törmääkö kaksi liikkuvaa palloa on laskea se niiden parametrisoiduista liikkeistä. Pallojen S_0 ja S_1 keskipisteiden sijainnit tietyllä ajanhetkellä voidaan ilmaista seuraavilla funktioilla: $P_0(t) = \mathbf{c}_0 + t\mathbf{v}_0$ ja $P_0(t) = \mathbf{c}_0 + t\mathbf{v}_0$, missä $0 \leq t \leq 1$. Pallojen keskipisteiden välinen vektori tietyllä ajanhetkellä saadaan kaavalla 23.

$$\mathbf{d}(t) = (\mathbf{c}_0 + t\mathbf{v}_0) - (\mathbf{c}_1 + t\mathbf{v}_1) = (\mathbf{c}_0 - \mathbf{c}_1) + t(\mathbf{v}_0 - \mathbf{v}_1) \tag{23}$$

Jos pallot eivät alussa törmää, niin ne törmäävät hetkenä, jolloin

$$\mathbf{d}(t) \bullet \mathbf{d}(t) = (r_0 + r_1)^2. \tag{24}$$

Ratkaisemalla yhtälöstä 24 t saadaan hetki, jolloin pallot törmäävät. Merkitään $\mathbf{s} = \mathbf{c}_0 - \mathbf{c}_1$, $\mathbf{v} = \mathbf{v}_0 - \mathbf{v}_1$ ja $r = r_0 + r_1$. Tästä saadaan yhtälö 25.

$$\begin{aligned} &\mathbf{d}(t) \bullet \mathbf{d}(t) = (r_0 + r_1)^2 \\ &\Rightarrow (\mathbf{s} + t\mathbf{v}) \bullet (\mathbf{s} + t\mathbf{v}) = r^2 \\ &\Rightarrow (\mathbf{v} \bullet \mathbf{v})t^2 + 2(\mathbf{v} \bullet \mathbf{s})t + \mathbf{s} \bullet \mathbf{s} - r^2 = 0 \end{aligned} \tag{25}$$

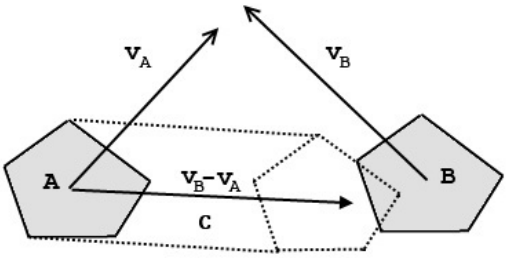
Tulos on toisen asteen yhtälö, missä $a = \mathbf{v} \bullet \mathbf{v}$, $b = 2(\mathbf{v} \bullet \mathbf{s})$ ja $c = \mathbf{s} \bullet \mathbf{s} - r^2$, joten t:n ratkaisuksi saadaan

$$t = \frac{-b \pm \sqrt{d}}{2a}, \text{missä } d = b^2 - 4ac \tag{26}$$

Jos $d < 0$, niin yhtään reaalijuurta ei ole olemassa. Tällöin pallot eivät törmää.
Jos $d > 0$, niin reaalijuuria on kaksi. Tällöin pienempi t on hetki, jolloin pallot törmäävät ja suurempi t on hetki, jolloin pallot eroavat. Jos $d = 0$ koskettavat pallot toisiaan niin läheltä, että pallot osuvat ja eroavat samasta pisteestä. Jos $a = 0$ kulkevat pallot eri suuntiin. Tämän laskentatavan yhteydessä on muistettava ensin tarkistaa törmäävätkö pallot jo alussa, muuten saatu tulos on virheellinen. [1, s. 223 – 226.]

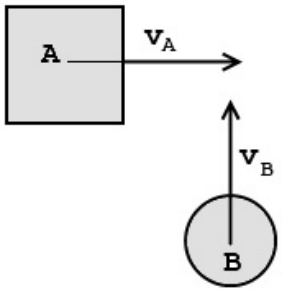
6.2 GJK ja liike

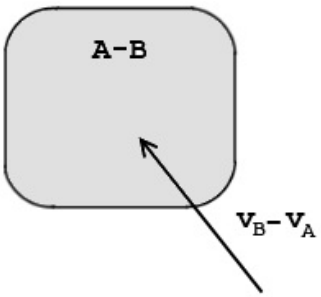
Helpoin tapa havaita GJK-algoritmilla törmääkö kaksi liikkuvaa kappaletta A ja B, on vähentää kappaleen A liike kappaleen B liikkeestä. Nyt kappale B on paikallaan oleva kappale ja A:n liikkeeksi asetetaan $\mathbf{v}_B - \mathbf{v}_A$. Seuraavaksi siirretään kappale A uuteen sijaintiin. Muodostetaan kappale C, joka rakentuu A:n pisteistä ennen ja jälkeen siirtoa kuvan 45 osoittamalla tavalla, jossa katkoviivat esittävät A:n pyyhkäisemää tilavuutta. Tämän jälkeen voidaan tarkistaa tavallisella GJK-algoritmilla törmäävätkö kappaleet C ja B, koska ongelma on saatu muotoon törmääkö kaksi paikallaan olevaa kappaletta. [1].



KUVA 45. Pyyhkäisevä tilavuus

Ongelma tässä menetelmässä on hitaus, jos kappale sisältää useita pisteitä. Yhden pisteen siirto vaatii 16 kertolaskua ja 16 yhteenlaskua. Käytännössä on hyvinkin mahdollista, että kappale koostuu tuhansista pisteistä, joten tämä menetelmä saattaa olla hidas. Parempi tapa havaita törmäävätkö kappaleet A ja B on tarkistaa onko mikään janan $\mathbf{v}_B - \mathbf{v}_A$ pisteistä A:n ja B:n Minkowskin erotuksen sisällä kuvien 45 ja 46 osoittamalla tavalla.



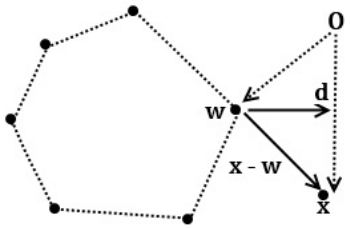


KUVA 46. Liikkuvat kappaleet [9] **KUVA 47. Jana ja Minkowskin erotus [9]**

Seuraava algoritmi perustuu Van Bergenin artikkeliin[9] ja esitykseen [8]. Tätä varten joudutaan tekemään GJK-algoritmiin seuraavat muutokset: Aiemmin esitellyssä GJK:ssa tutkittiin onko origo Minkowskin erotuksen sisällä. Nyt täytyy tutkia onko tietty piste Minkowskin erotuksen sisällä. Lisäksi joudutaan tekemään muutos, jossa tutkitaan, mikä on tämän pisteen etäisyys simpleksiin. Jos pisteen etäisyys simpleksistä on alle tietyn virhemarginaalin, niin se tulkitaan olevan Minkowskin erotuksen sisällä. Ensin käydään läpi kuinka muutokset aiemmin esiteltyyn GJK-algoritmiin tehdään. Tämän jälkeen esitellään tapa, kuinka janasta $\mathbf{v}_B - \mathbf{v}_A$ lasketaan piste, joka tarkistetaan onko se Minkowskin erotuksen sisällä. Lopuksi yhdistetään nämä kaksi.

6.2.1 Muutokset GJK-algoritmiin

Algoritmin ajatus on edelleen sama, eli pyritään sulkemaan piste kartion sisälle. Tämä ei ole välttämätön ehto lopetukseen, koska lopetusehto on simpleksin etäisyys pisteestä. Algoritmi etenee seuraavasti: Valitaan mikä tahansa piste **v** Minkowskin erotuksesta, mutta ei lisätä tätä simpleksiin. Asetetaan etsintäsuunnaksi $\mathbf{d} = \mathbf{x} - \mathbf{v}$, missä **x** on piste, joka tutkitaan onko se Minkowskin erotuksen sisällä. Etsitään etäisin piste **w** suunnasta **d**. Jos $\mathbf{d} \bullet (\mathbf{x} - \mathbf{w}) > 0$, niin **x** ei ole Minkowskin erotuksen sisällä, kuva 48 havainnollistaa tämän.



KUVA 48. x Minkowskin erotuksen ulkopuolella

Muuten lisätään piste **w** simpleksiin. Seuraavaksi kutsutaan alialgoritmia. Alialgoritmia on muutettu niin, että se palauttaa lähimmän pisteen **p** simpleksistä pisteeseen **x**. Lopuksi asetetaan etsintäsuunnaksi **d** = **x** – **p**. Tätä toistetaan, kunnes **d**:n etäisyys pisteeseen **x** on alle tietyn virhemarginaalin ε. Pääalgoritmin tarkoitus on myös tässä vain lisätä uusi piste simpleksiin ja palauttaa *törmäys* tai *ei törmäystä*. Pseudokoodi 9 esittää muutetun GJK-algoritmin pääalgoritmia.

Pseudokoodi 9. GJK-algoritmi 2.

```
Simpleksi on tyhjä taulukko
x = Testattava piste
v = Mikä tahansa piste Minkowskin erotuksesta
d = x - v
p = 0

while ||d||^2 < ε
    w = S_A-B(d)

    if d • (x – w) > 0
        Lopeta palauta ei törmäystä
    end if

    Lisää w simpleksiin
    Kutsu alialgoritmia
    d = x - p
end while

Palauta törmäys
```

6.2.2 Muutokset alialgoritmiin

Janan käsittely on muutettu niin, että se ainoastaan laskee lähimmän pisteen janasta **ab** pisteeseen **x**, ja asettaa sen **p**:n arvoksi kohdassa 3 esitellyllä tavalla.

Kolmion käsittelyyn on tullut eniten muutoksia. Sivun **ac** ollessa lähin pisteeseen **x**, asetetaan simpleksiksi **c**, **a** ja siirrytään janan käsittelyyn (kuva 29). Janan käsittelyn yhteydessä asetetaan lähin piste, joten sitä ei tässä tarvitse tehdä. Myös tässä yhteydessä **a**:lla tarkoitetaan viimeksi lisättyä pistettä. Sivun **ab** ollessa lähinnä asetetaan simpleksiksi **b**, **a** ja siirrytään janan käsittelyyn.

Jos kumpikaan sivuista ei ole lähinnä pistettä **x**, tarkistetaan kummalla puolella kolmionpintaa **x** on. Jos **x** on suunnassa jonne **abc** osoittaa, asetetaan simpleksiksi **b**, **c**, **a** ja lasketaan mikä piste kolmiosta on lähinnä pistettä **x** ja asetetaan se pisteeksi **p**. Jos **x**

on suunnassa, jonne –**abc** osoittaa, lasketaan vain lähin piste. Lähimmän pisteen laskussa on hyvä huomata, että tämä voidaan laskea mikä piste on pisteiden **a, b** ja **c** muodostamassa tasossa lähinnä, koska tiedetään, että piste on kolmion sisällä. Tämä voidaan tehdä luvussa 3 esitellyllä tavalla. Pseudokoodissa 10 on esitetty kolmion käsittely.

Pseudokoodi 10. GJK-algoritmi 2 kolmion käsittely.

```
ab = b - a
ac = c – a
ax = x - a
abc = ab × ac

if (abc × ac) • ax > 0
    Simpleksi on [c, a]
    Siirry janan käsittelyyn
end if

if (ab × abc) • ax > 0
    Simpleksi on [b, a]
    Siirry janan käsittelyyn
end if

if abc • ax > 0
    Simpleksi on [b, c, a]
    p = Lähin piste tasosta pisteeseen x
else
    p = Lähin piste tasosta pisteeseen x
end if
```

Kartion käsittelyyn ei ole tehty kuin kaksi muutosta. Ensimmäinen on se, että ei tarkisteta onko origo kolmion sisällä, vaan tarkistetaan onko **x** kartion sisällä. Toinen muutos on, että jos **x** on kartion sisällä, asetetaan pisteeksi **p** = **x**, koska lähin piste pisteeseen **x** on **x**. Tämä tietysti aiheuttaa sen, että pääalgoritmi päättyy, koska $\|\mathbf{x} - \mathbf{x}\| = 0$. Pseudokoodissa 11 on esitetty kartion käsittely.

Pseudokoodi 11. GJK-algoritmi 2, kartion käsittely.

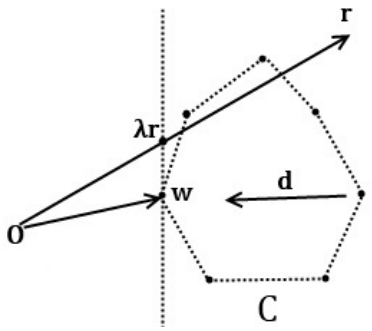
```
ax = x -a

bac = (a – b) × (c – b)
if bac • ax > 0
    Simpleksi on [c, b, a]
    Siirry kolmion käsittelyyn
end if

cad = (a – c) × (d – c)
if cad • ax > 0
    Simpleksi on [d, c, a]
    Siirry kolmion käsittelyyn
end if

dab = (a – d) × (b – d)
if dab • ax > 0
    Simpleksi on [b, d, a]
    Siirry kolmion käsittelyyn
end if

p = x.
```

6.2.3 Tarkistettavan pisteen laskeminen janasta

Määritellään konveksin kappaleen C:n pinnan normaali **d** pisteessä **w** kaavalla 27.

$$\mathbf{d} \bullet (\mathbf{x} - \mathbf{w}) \leq 0, \text{ jokaiselle } \mathbf{x} \in C \tag{27}$$

Tämä tarkoittaa sitä, että **d** on sellaisen tason normaali, joka kulkee pisteen **w** kautta ja jonka taakse kaikki C:n pisteet jäävät. Tarkastellaan kuvan 49 tilannetta, jossa tarkistetaan osuuko säde **r** kappaleeseen C.



KUVA 49. Uuden pisteen laskeminen janasta

Alkutilanteessa tutkitaan, onko janan alkupiste eli origo kappaleen sisällä. Kuvan 49 tilanteessa on etsitty etäisin piste **d**:n suunnasta ja löydetty piste **w**. Nyt havaitaan, että origo ei ole Minkowskin erotuksen sisällä, koska $\mathbf{w} \bullet (\mathbf{x}_0 - \mathbf{d}) > 0$, missä $\mathbf{x}_0$ on origo. Nyt **w** on piste konveksin kappaleen pinnalta ja **d** on tämän pinnan normaali pisteessä **w**. Seuraavaksi lasketaan $\mathbf{x}_1$ janasta **r** kaavalla 28.

$$\mathbf{x} = \lambda \mathbf{r} \tag{28}$$

Kaavassa 28 oleva λ saadaan merkitsemällä yhtälö 27 yhtä suureksi kuin nolla ja ratkaisemalla λ, jolloin saadaan kaava 29.

$$\lambda = \frac{\mathbf{d} \bullet \mathbf{w}}{\mathbf{d} \bullet \mathbf{r}} \tag{29}$$

Saatu $\mathbf{x}_{i+1}$ on piste janasta **r**, joka osuu tasoon ja jonka normaali on **d**. Taso kulkee pisteen **w** kautta, kuvassa 49 katkoviiva esittää tätä tasoa.

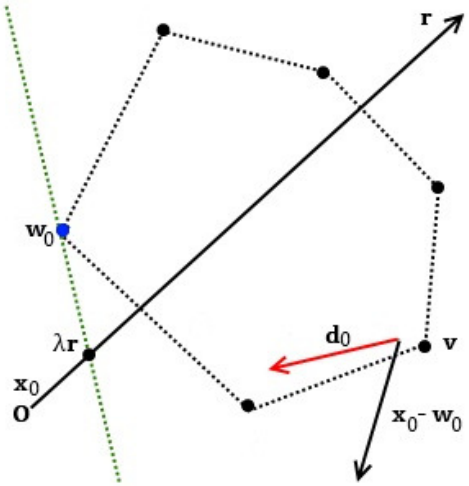
6.2.4 Yhdistetty algoritmi

Algoritmien yhdistäminen vaati vain sen, että tehdään muutos kohtaan, jossa tarkistetaan onko $\mathbf{d} \bullet (\mathbf{x} - \mathbf{w}) > 0$. Jos $\mathbf{d} \bullet (\mathbf{x} - \mathbf{w}) > 0$ niin tarkistetaan onko $\mathbf{d} \bullet \mathbf{r} \geq 0$, jolloin säde suuntautuu kappaleesta poispäin. Näin ollen mikään piste säteestä **r** ei voi olla Minkowskin erotuksen sisällä. Jos $\mathbf{d} \bullet \mathbf{r} < 0$ lasketaan uusi λ arvo kaavalla 29. Jos $\lambda > 1$, niin jana **r** ei osu kappaleeseen, joten palautetaan, *ei törmäystä*. Muuten lasketaan uusi **x**. Joka kerta kun uusi vektori **x** lasketaan, asetetaan simpleksiksi viimeksi haettu piste, josta lähdetään uutta simpleksiä rakentamaan. Pseudokoodi 12 esittää yhdistettyä algoritmia.

Pseudokoodi 12. GJK-algoritmi ja säde.

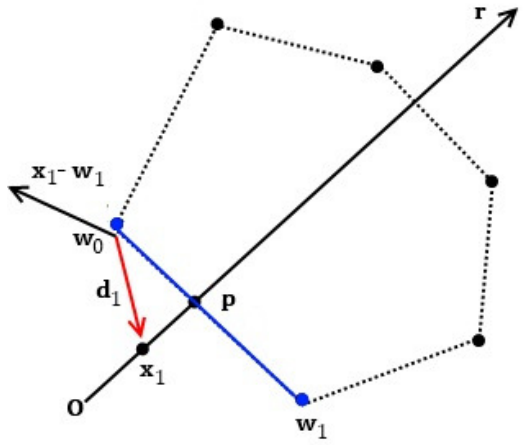
```
Simpleksi on tyhjä taulukko
r = v_B - v_A
x = Origo
v = Mikä tahansa piste Minkowskin erotuksesta
d = x - v
p = 0

while ||d||^2 < ε
    w = S_A-B(d)

    if d • (x – w) > 0
        if d • r ≥ 0
            Palauta ei törmäystä
        end if

        λ = (d • w) / (d • r)

        if λ > 1
            Palauta ei törmäystä
        end if

        x = λr
        Tyhjennä simpleksi
    end if

    Lisää w simpleksiin
    Kutsu alialgoritmia
    d = x - p
end while

Palauta törmäys
```

Seuraava kuvasarja esittää algoritmin toiminnan kaksiulotteisessa avaruudessa. Kuva 50 esittää alkutilannetta, jossa $\mathbf{x}_0$ on origo. Alkutilanteessa on valittu mielivaltainen piste $\mathbf{v}$ Minkowskin erotuksesta ja asetettu etsintäsuunnaksi $\mathbf{d}_0 = -\mathbf{v}$. Suunnasta $\mathbf{d}_0$ etäisin piste on $\mathbf{w}_0$. Koska $\mathbf{d}_0 \bullet (\mathbf{x}_0 - \mathbf{w}_0) > 0$, lasketaan uusi $\mathbf{x}_1$ ja asetetaan simpleksi, viimeksi haettuun pisteeseen $\mathbf{w}_0$. Lasketaan lähin piste $\mathbf{p}_0$ simpleksistä pisteeseen $\mathbf{x}_1$, joka on piste $\mathbf{w}_0$, koska simpleksi sisältää vain yhden pisteen. Asetetaan seuraavaksi etsintäsuunnaksi $\mathbf{d}_1 = \mathbf{x}_1 - \mathbf{p}_0$.



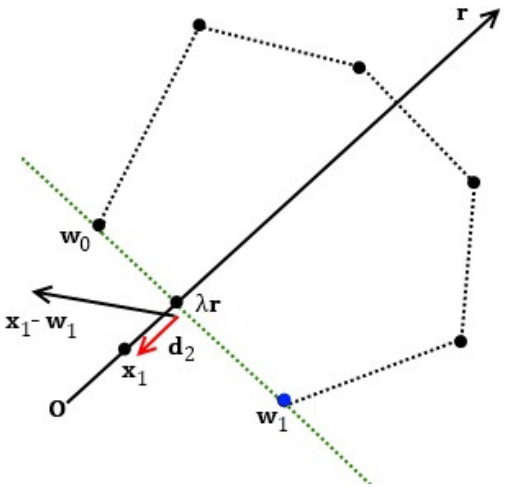
KUVA 50. Vaihe 1

Kuvassa 51 suunnassa $\mathbf{d}_1$ etäisin piste on $\mathbf{w}_1$, koska $\mathbf{d}_1 \bullet (\mathbf{x}_1 - \mathbf{w}_1) \leq 0$. Uutta $\mathbf{x}$:n arvoa ei lasketa eikä simpleksille tehdä mitään. Lisätään $\mathbf{w}_1$ simpleksiin ja lasketaan lähin piste $\mathbf{p}_1$ simpleksistä pisteeseen $\mathbf{x}_1$. Asetetaan seuraavaksi etsintäsuunnaksi $\mathbf{d}_2 = \mathbf{x}_1 - \mathbf{p}_1$.



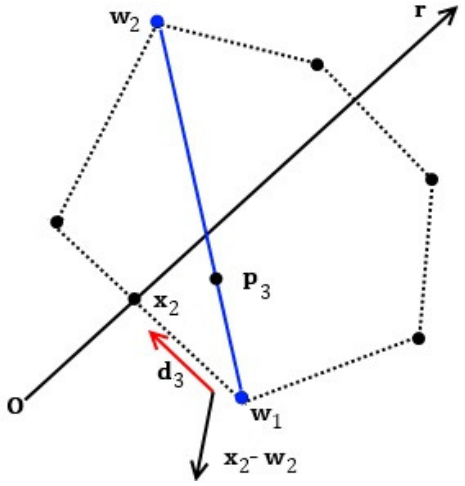
KUVA 51. Vaihe 2

Kuvassa 52 suunnasta $\mathbf{d}_2$ etäisin piste voi olla $\mathbf{w}_0$ tai $\mathbf{w}_1$. Valitaan tämän esimerkin kannalta $\mathbf{w}_1$. Koska $\mathbf{d}_2 \bullet (\mathbf{x}_1 - \mathbf{w}_1) > 0$, lasketaan uusi $\mathbf{x}_2$. Asetetaan simpleksi viimeksi haettuun pisteeseen, eli pisteeseen $\mathbf{w}_1$. Lasketaan simpleksistä lähin piste $\mathbf{p}_2$, joka on $\mathbf{w}_1$, koska simpleksi sisältää vain yhden pisteen. Asetetaan seuraavaksi etsintäsuunnaksi $\mathbf{d}_3 = \mathbf{x}_2 - \mathbf{p}_2$.



KUVA 52. Vaihe 3

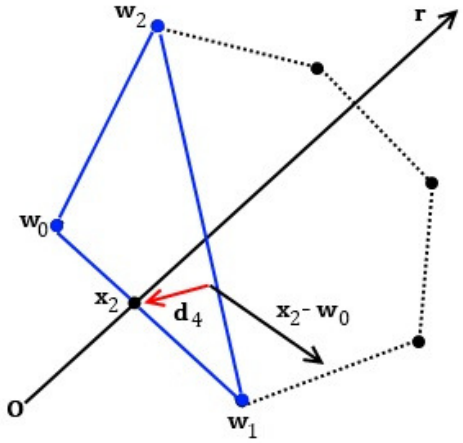
Kuvassa 53 suunnassa $\mathbf{d}_3$ etäisin piste on $\mathbf{w}_2$. Koska $\mathbf{d}_3 \bullet (\mathbf{x}_2 - \mathbf{w}_2) \leq 0$, uutta $\mathbf{x}$:n arvoa ei lasketa eikä simpleksille tehdä mitään. Simpleksin rakentamista jatketaan lisäämällä $\mathbf{w}_2$ simpleksiin. Lasketaan lähin piste $\mathbf{p}_3$ simpleksistä pisteeseen $\mathbf{x}_2$ ja asetetaan seuraavaksi etsintäsuunnaksi $\mathbf{d}_4 = \mathbf{x}_2 - \mathbf{p}_3$.



KUVA 53. Vaihe 4

Kuvassa 54 suunnassa $\mathbf{d}_4$ etäisin piste on piste $\mathbf{w}_0$. Koska $\mathbf{d}_4 \bullet (\mathbf{x}_2 - \mathbf{w}_0) \leq 0$, uutta $\mathbf{x}$:n arvoa ei lasketa eikä simpleksille tehdä mitään. Lisätään piste $\mathbf{w}_0$ simpleksiin. Lasketaan lähin piste $\mathbf{p}_4$ simpleksistä pisteeseen $\mathbf{x}_2$, joka on $\mathbf{x}_2$. Asetetaan seuraavaksi etsintäsuunnaksi $\mathbf{d}_5 = \mathbf{x}_2 - \mathbf{p}_4$. Nyt havaitaan, että $\|\mathbf{d}_5\|^2 < \varepsilon$, joten palautetaan *törmäys*. Tästä havaitaan virhemarginaalin tärkeys. Minkowskin erotuksen sisälle ei päästä, päästään vain pinnalle ja liukulukujen kanssa nollaan vertaaminen on riskialtista, joka voi aiheuttaa ikuisen silmukan. Se, että Minkowskin erotuksen sisälle ei päästä, on

hyödyllinen ominaisuus, koska nyt voidaan λ arvon perusteella siirtää kappaleet niin, että ne juuri koskettavat.



KUVA 54. Vaihe 5

Van Bergen ehdotti myös seuraavaa lopetusehtoa:

$$\frac{\|\mathbf{x}-\mathbf{p}\|^2}{\max\{\|\mathbf{x}-\mathbf{y}\|^2 : \mathbf{y} \in W\}} \leq \varepsilon \tag{30}$$

Kaavassa 30 W on simpleksi ja **y** on piste simpleksissä. Käytännössä havaitsin tämän lopetusehdon huomattavasti luotettavammaksi kuin verrata **p**:n etäisyyttä suoraan pisteeseen **x**. Virhemarginaalin ε valinta vaikuttaa myös huomattavasti suorituskykyyn. Van Bergenin kokeiden mukaan kun $\varepsilon = 10^{-6}$, aikaa kului keskimäärin noin 2,8 kertaa enemmän kuin tilanteessa $\varepsilon = 10^{-3}$.[8][9].

7 YHTEENVETO JA JATKOKEHITYS

Tässä luvussa kerrataan lyhyesti mitä tässä työssä käsiteltiin sekä esitetään mahdollisia jatkokehitysideoita.

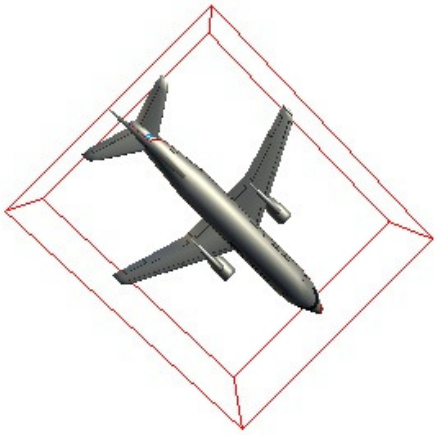
7.1 Ympäröivät peitteet

Työn alkuosassa luvuissa 3 ja 4 käsiteltiin ympäröiviä peitteitä. Törmäyksen tunnistuksen helpottamiseksi ja nopeuttamiseksi voidaan mallin tai pistejoukon ympärille määrätä jokin tunnettu geometrinen kappale. Useasti käytännössä mallin tai pistejoukon ympärille määrätään useampia ympäröiviä peitteitä, jotta voidaan ensin tehdä

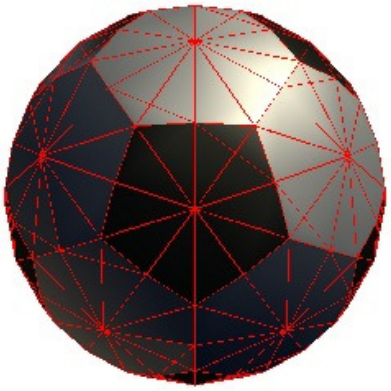
törmäyksen tunnistus yksinkertaisemmalla ja vähemmän laskentatehoa vaativalla peitteellä esim. pallolla. Jos havaitaan törmäys, siirrytään tarkempaan ja enemmän laskentatehoa vaativaan törmäyksen tunnistukseen. Törmäyksen tunnistamisen helpottamiseksi on paljon muitakin menetelmiä, kuten ympäröivistä peitteistä koostuvat hierarkiat, avaruuden jako jne. Avaruuden jaossa avaruus jaetaan rekursiivisesti osiin, jotta voidaan pois sulkea kappaleet mihin ei voida törmätä tähän ja moniin muihin menetelmiin Ericson [3] tarjoaa kattavan johdannon.

Tässä työssä ympäröivistä peitteistä käsiteltiin orientoitunut laatikko ja konveksi peite. Orientoituneen laatikon määräämisessä tavoite on saada mahdollisimman pieni laatikko, joka sulkee pistejoukon sen sisälleen. Pienimmän laatikon löytäminen on haastava ongelma ja tätä havainnollistaa se, että vasta vuonna 1985 O'Rourke[12] julkaisi ensimmäisenä algoritmin, joka määrää mielivaltaisen pistejoukon ympärille pienimmän mahdollisen laatikon.

Orientoituneen laatikon määräsin tässä työssä pelkkien mallin pisteiden perusteella luvussa 3 esitetyllä tavalla ja sain mielestäni kelpo tuloksia. Kuva 55 esittää komiulotteisen mallin ympärille määrättyä orientoitunutta laatikkoa. Kuva 56 esittää mallin ympärille määrättyä konveksia peitettä.



KUVA 55. Orientoitunut laatikko



KUVA 56. Konveksi peite

7.2 Törmäyksen tunnistusalgoritmit

Tässä työssä esitellyt törmäyksen tunnistus algoritmit keskittyivät konvekseihin kappaleisiin. Pääpainopisteenä oli GJK- ja Chung-Wang-algoritmit. Algoritmit muistutta-

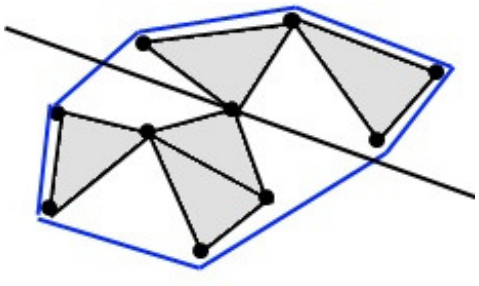
vat toisiaan siinä mielessä, että molemmat perustuvat Minkowskin erotukseen. Molemmissa algoritmeissa yksi raskaimmista laskutoimituksista on laskea etäisin piste suunnasta. Mikäli kappale on esitetty pistejoukkona, joudutaan iteroimaan läpi kaikki pisteet ennen kuin tiedetään mikä on etäisin piste suunnassa (kaava 13).

Tätä voidaan nopeuttaa kiipeilyalgoritmilla. Kiipeilyalgoritmin ajatus on yksinkertainen. Valitaan mielivaltainen kärkipiste. Tarkistetaan, onko tämä etäämpänä etsintäsuunnassa kuin mikään pisteen naapuripisteistä. Jos piste on etäämpänä kuin mikään sen naapuripisteistä, niin piste on etäisin suunnassa, koska kappale on konveksi. Muuten valitaan etäisin naapuripiste, siirrytään siihen ja tarkistetaan onko se etäämpänä kuin mikään sen naapureista. Tätä toistetaan, kunnes saavutaan pisteeseen, joka on etäämpänä kuin mikään sen naapureista. Luvussa 4 esitetty konveksi peite algoritmi vaati vain pieniä muutoksia, jotta jokaisesta pisteestä saadaan tieto sen naapuripisteistä.

Molemmat Chung-Wang-algoritmi sekä GJK-algorimi havaitsevat törmäyksen ainoastaan konvekseilla kappaleilla. Mikäli niitä käytetään ei-konvekseilla kappaleilla, törmäys havaitaan kappaleiden konveksipeitteistä.

7.3 Ei-konveksit kappaleet

Ei-konveksien kappaleiden käsittelyssä yksi menetelmä on pilkkoa kappale useampaan konveksiin kappaleeseen, jotta voidaan käyttää konveksisuteen perustuvia menetelmiä. Konveksiin osiointi on hankala ongelma 3D ympäristössä ja tässä annetaan lyhyt kuvaus. Rakennetaan pistejoukon ympärille konveksipeite. Etsitään tahkot, jotka ovat aukkojen päällä. Valitaan joku näistä tahkoista, etsitään etäisin näkyvä piste tästä tahkosta ja leikataan kappale kahtia tämän pisteen kautta kuvan 57 osoittamalla tavalla.



KUVA 57. Leikkaava taso

Ensimmäinen haaste tässä menetelmässä on se, kuinka lasketaan etäisin näkyvä piste tahkosta. Ei voida vain huomioida pisteitä vaan täytyy myös ottaa huomioon kuinka ne yhdistyvät toisiinsa ts. täytyy etsiä etäisin näkyvä piste polygoneista. Toinen ongelma on kuinka valita leikkaava taso.

Toinen huomattavasti helpompi menetelmä konveksiin osiointiin on ns. tilavuusmenetelmä. Idea on yksinkertainen. Ryhmitellään mallin kolmiot pieniin ryhmiin. Lasketaan jokaisen ryhmän ympärille, jokin ympäröivätilavuus esim. laatikko. Seuraavaksi pyritään etsimään kaksi laatikkoa, joiden ympärille lasketun laatikon tilavuus on lähellä näiden kahden laatikon tilavuuden summaa. Jos kaksi tällaista laatikkoa löytyi, yhdistetään ne. Toistetaan tätä niin kauan, kunnes on saatu haluttu määrä konvekseja kappaleita. Tämä on suhteellisen helppo toteuttaa ja kokeilin tätä menetelmää käyttäen koordinaatiston mukaan orientoituneita laatikoita. Tein ryhmittelyn koordinaatiston mukaan orientoituneilla laatikoilla ja lopuksi laskin ryhmien ympärille orientoituneet laatikot Kuvat 58 ja 59 esittävät kahta mallia, joiden kanssa tämä menetelmä tuotti kohtuullisen hyvän lopputuloksen. Ratcliff[11] on kehitellyt tätä muistuttavaa menetelmää ja tulokset ovat kuvien perusteella olleet erittäin hyviä.

KUVA 58. Konveksiin osioitu portti **KUVA 59. Konveksiin osioitu lentokone**

7.4 Etäisin piste kappaleesta ja törmäysimpulssit

Törmäyksen tunnistuksen viimeinen vaihe on törmäysimpulssien laskeminen. Ennen kuin törmäysimpulssit voidaan laskea, täytyy tietää mitkä kohdat kappaleista törmää. GJK-algoritmissa törmäyskohtien selvittämiseksi voidaan hyödyntää pisteitä, jotka muodostivat Minkowskin erotuksen. Etsimällä kappaleista osat, joihin nämä pisteet kuuluvat, saadaan potentiaaliset törmäyskohdat. Tämän menetelmän luotettavuus riip-

puu siitä, kuinka paljon kappaleet lävistävät toisensa. Mitä enemmän kappaleet lävistävät toisensa, sitä epäluotettavampi tämä menetelmä on.

Törmäyskohtien selvittämiseen liittyy myös olennaisesti etäisimmän pisteen haku. Etäisimmän pisteen hakuun konveksista monitahokkaasta O'Rourke[6] tarjoaa mielenkiintoisen algoritmin, joka on suhteellisen helppo viedä käytäntöön. Algoritmi perustuu graafiteorian itsenäiseen sarjaan. Algoritmin idea on seuraavanlainen: Etsitään monitahokkaasta itsenäiset pisteet, poistetaan ne ja lasketaan jäljelle jääneiden pisteiden ympärille konveksipeite. Jatketaan tätä kunnes on saavuttu kolmioon tai kartioon.

Etäisimmän pisteen haku tietystä suunnasta aloitetaan etsimällä sisimmästä kappaleesta, joka on kartio tai kolmio, etäisin piste. Seuraavaksi siirrytään monitahokkaaseen, josta poistetut itsenäiset pisteet tuottivat kolmion tai kartion. Etäisin piste tästä monitahokkaasta on joko sama piste tai joku tämän pisteen naapuripisteistä, jotka poistettiin. Algoritmi tavallaan kiipeilee kohti ulointa monitahokasta.

LÄHTEET

1. Ericson Christer. Real-Time Collision Detection. Morgan Kaufmann Publishers. 2005.

2. Gottschalk, Stefan. Collision Queries using Oriented Bounding Boxes. Phd. Thesis. University of North Carolina at Chapel Hill, Department of Computer Science. 2000. http://gamma.cs.unc.edu/users/gottschalk/main.pdf. Päivitetty 16.5.2001.

3. O'Rourke, Joseph. Computational Geometry in C, second edition. Cambridge University Press. 1998.

4. Muratori, Casey. Implementing GJK. Videoesitys. 2006. Saatavissa: https://mollyrocket.com/849

5. Chung, Kelvin. An Efficient Collision Detection Algorithm for Polytopes in Virtual Environments. Masters Thesis. University of Hong Kong, Department of Computer Science. 1996. http://i.cs.hku.hk/GraphicsGroup/publications/thesis/Tat-Leung Chung - master.pdf. Päivitetty 23.12.2005.

6. Chung, Kelvin ja Wenping, Wang. Quick Collision Detection of Polytopes in Virtual Environments. 1996. http://i.cs.hku.hk/GraphicsGroup/projects/collision/cw_vrst96.pdf. Päivitetty 16.12.2002.

7. Van Den Bergen, Gino. Collision Detection in Interactive 3D Computer Animation. Phd. Thesis. Eindhoven University of Technology. 1999. http://alexandria.tue.nl/extra2/9900696.pdf. Päivitetty 8.3.2000.

8. Van Den Bergen, Gino. Continuous Collision Detection. 2004. http://homepages.laas.fr/nic/MOVIE/Workshop/Slides/Gino.vander.Bergen.ppt. Päivitetty 20.12.2004.

9. Van Den Bergen, Gino. Ray Casting against General Convex Objects with Application to Continuous CollisionDetection. 2004. http://www.bulletphysics.com/ftp/pub/test/physics/papers/jgt04raycast.pdf. Päivitetty 15.6.2004.

10. Van Den Bergen, Gino. A Fast and Robust GJK Implementation for Collision Detection of Convex Objects. 1999. http://www.win.tue.nl/~gino/solid/jgt98convex.pdf. Päivitetty 6.7.1999.

11. Ratcliff, John. Approximate ConvexDecomposition. http://codesuppository.blogspot.com/2006/08/approximate-convexdecomposition.html

12. O'Rourke, Joseph. Finding Minimal Enclosing Boxes. 1985. http://maven.smith.edu/~orourke/Papers/MinVolBox.pdf. Päivitetty 16.12.2010.

```
public class Vertex
{
    private Vector3 position;
    private Edge newEdge = null;
    private Vector3 originalPosition;

    public void Transform(Matrix worldMatrix)
    {
       position = Vector3.Transform(originalPosition, worldMatrix);
    }
}
public class TriangleFace
{
    public Edge[] Edge { get; set; }
    public Vertex[] Vertices { get; set; }
    public bool Visible { get; set; }

    public TriangleFace(Edge edge0, Edge edge1, Edge edge2): base()
    {
       Visible = false;
       Edge = new Edge[3];
       Vertices = new Vertex[3];
       Edge[0] = edge0; Edge[1] = edge1; Edge[2] = edge2;
     }

     public TriangleFace(Edge edge0, Edge edge1, Edge edge2,
                    Vertex v0, Vertex v1, Vertex v2)
      : base(){
       Edge = new Edge[3];
       Vertices = new Vertex[3];
       Visible = false;
       Vertices[0] = v0; Vertices[1] = v1; Vertices[2] = v2;
       Edge[0] = edge0; Edge[1] = edge1; Edge[2] = edge2;
     }
}
public class Edge
{
    public TriangleFace AdjacentFace1 { get; set; }
    public TriangleFace AdjacentFace2 { get; set; }
    public Vertex EndPoint1 { get; set; }
    public Vertex EndPoint2 { get; set; }
    public TriangleFace NewFace { get; set; }

    public Edge(Vertex endPoint1, Vertex endPoint2)
    {
       this.EndPoint1 = endPoint1;
       this.EndPoint2 = endPoint2;
    }

    public bool IsBorderEdge()
    {
        int visibleFaces = 0;

        if (AdjacentFace1.Visible)
            visibleFaces++;

        if (AdjacentFace2.Visible)
            visibleFaces++;

        if (visibleFaces == 1)
            return true;
        else
            return false;
    }
}
```

Perustuu lähteeseen [3].

Konveksipeitteen algoritmi

```
private void MakeConvexHull(List<Vertex> vertices){
    faces = new List<IFace>();
    // Haetaan 3 pistettä, jotka muodostavat kolmion
    Vertex[] trienglePoints = TreeNonCollinearPoints(vertices);
    // Rakennetaan kaksi vastakkain kierrettyä kolmiota
    Edge edge1 = new Edge(trienglePoints[0], trienglePoints[1]);
    Edge edge2 = new Edge(trienglePoints[1], trienglePoints[2]);
    Edge edge3 = new Edge(trienglePoints[2], trienglePoints[0]);

    TriangleFace face1 = new TriangleFace(edge1, edge2, edge3,
          trienglePoints[2], trienglePoints[1], trienglePoints[0]);

    TriangleFace face2 = new TriangleFace(edge1, edge2, edge3,
          trienglePoints[0], trienglePoints[1], trienglePoints[2]);

    edge1.AdjacentFace1 = face1;
    edge1.AdjacentFace2 = face2;
    edge2.AdjacentFace1 = face1;
    edge2.AdjacentFace2 = face2;
    edge3.AdjacentFace1 = face1;
    edge3.AdjacentFace2 = face2;
    faces.Add(face1);
    faces.Add(face2);

    // Käydään läpi jokainen piste
    for (int i = 0; i < vertices.Count; i++)
    {
        // Merkitään näkyvät tahkot
        bool visibleFacesExists =
                        MarkVisibleFacesFromVertex(vertices[i], faces);
        // Jos näkyviä tahkoja löytyi käydään läpi tahkot ja
           etsitään näkyvät tahkot
        if (visibleFacesExists)
        {
            int facesCount = faces.Count;
            for (int j = 0; j < facesCount; j++)
            {
                // Onko tahko merkitty näkyväksi
                if (faces[j].Visible)
                {
                    // Rakennetaan näkyvän alueen särmistä uudet tahkot
                    for (int k = 0; k < 3; k++)
                    {
                        Edge edge = faces[j].Edge[k];
                        if (edge.IsBorderEdge())
                        {
                            TriangleFace newFace =
                                            MakeNewFace(edge, vertices[i]);
                            faces.Add(newFace);
                        }
                    }// for k
                    // Poistetaan käsitelty näkyvä tahko
                    faces.RemoveAt(j);
                    j--;
                    facesCount--;
                }
            }// for j
        }
    }// for i
}
```

Perustuu lähteeseen [3].

```
private TriangleFace MakeNewFace(Edge borderEdge, Vertex pointToAdd){
    Edge newEdge1 = borderEdge.EndPoint1.NewEdge;
    Edge newEdge2 = borderEdge.EndPoint2.NewEdge;

    if (newEdge1 == null){
        newEdge1 = new Edge(borderEdge.EndPoint1, pointToAdd);
        borderEdge.EndPoint1.NewEdge = newEdge1;
    }else
        borderEdge.EndPoint1.NewEdge = null;

    if (newEdge2 == null){
        newEdge2 = new Edge(borderEdge.EndPoint2, pointToAdd);
        borderEdge.EndPoint2.NewEdge = newEdge2;
    }else
        borderEdge.EndPoint2.NewEdge = null;

    TriangleFace newFace = new TriangleFace (borderEdge, newEdge1, newEdge2);

    TriangleFace visibleFaceOfBorder = null;

    if (borderEdge.AdjacentFace1.Visible)
        visibleFaceOfBorder = borderEdge.AdjacentFace1;
    else
        visibleFaceOfBorder = borderEdge.AdjacentFace2;
    // Haetaan piste näkyvästä tahkosta, joka on seuraavana 'borderEdge' särmän
    // ensimmäisestä päätepisteestä
    TriangleFace v = null;
    for (int i = 0; i < 3; i++){
        if (visibleFaceOfBorder.Vertices[i] == borderEdge.EndPoint1){
            v = visibleFaceOfBorder.Vertices[(i + 1) % 3];
            break;
        }
    }
    // Orientoidaan tahkon kärkipisteet vastapäivään
    if (v != borderEdge.EndPoint2){
        newFace.Vertices[0] = borderEdge.EndPoint2;
        newFace.Vertices[1] = borderEdge.EndPoint1;
    }else{
        newFace.Vertices[0] = borderEdge.EndPoint1;
        newFace.Vertices[1] = borderEdge.EndPoint2;
    }
    newFace.Vertices[2] = pointToAdd;
    // Päivitetään särmien tietoihin, että ne yhdistyvät luotuun tahkoon
    if (newEdge1.AdjacentFace1 == null)
        newEdge1.AdjacentFace1 = newFace;
    else
        newEdge1.AdjacentFace2 = newFace;

    if (newEdge2.AdjacentFace1 == null)
        newEdge2.AdjacentFace1 = newFace;
    else
        newEdge2.AdjacentFace2 = newFace;

    if (borderEdge.AdjacentFace1.Visible)
        borderEdge.AdjacentFace1 = newFace;
    else
        borderEdge.AdjacentFace2 = newFace;

    return newFace;
}
```

Perustuu lähteeseen [3].

Näkyvien tahkojen merkitseminen

```
private bool MarkVisibleFacesFromVertex(Vertex vertex, List<IFace> listOfFaces)
{
    bool visibleFacesExists = false;

    for (int i = 0; i < listOfFaces.Count; i++)
    {
        Vertex a = listOfFaces[i].Vertices[0];
        Vertex b = listOfFaces[i].Vertices[1];
        Vertex c = listOfFaces[i].Vertices[2];
        Vertex d = vertex;

        double ax = a.X - d.X;
        double ay = a.Y - d.Y;
        double az = a.Z - d.Z;

        double bx = b.X - d.X;
        double by = b.Y - d.Y;
        double bz = b.Z - d.Z;

        double cx = c.X - d.X;
        double cy = c.Y - d.Y;
        double cz = c.Z - d.Z;

        // Kaava 11
        double volume = ax * (by * cz - bz * cy)
            + ay * (bz * cx - bx * cz) + az * (bx * cy - by * cx);

        if (volume < 0)
        {
            listOfFaces[i].Visible = true;
            visibleFacesExists = true;
        }
        else
        {
            listOfFaces[i].Visible = false;
        }
    }

    return visibleFacesExists;
}
```

Perustuu lähteeseen [3].

Kolme pistettä, jotka muodostavat kolmion

```
private Vertex [] TreeNonCollinearPoints(List <Vertex> vertices)
{
    Vertex [] triangle = new Vertex [3];
    triangle[0] = vertices[0];
    vertices.RemoveAt(0);
    //
    // Etsitään toinen piste, joka muodostaa janan
    for (int i = 0; i < vertices.Count; i++)
    {
        Vertex v = vertices[i];
        float distance = Vector3.Distance(vertices[i].Position,
             triangle[0].Position);

        if (distance != 0)
        {
            triangle[1] = vertices[i];
            vertices.RemoveAt(i);
            break;
        }
    }
    //
    // Etsitään kolmas piste, joka muodostaa kolmion
    for (int i = 0; i < vertices.Count; i++)
    {
        Vector3 a = triangle[0].Position;
        Vector3 b = triangle[1].Position;
        Vector3 c = vertices[i].Position;

        Vector3 ab = b - a;
        Vector3 ac = c - a;
        Vector3 abc = Vector3.Cross(ab, ac);

        if (abc.LengthSquared() > 0)
        {
            triangle[2] = vertices[i];
            break;
        }
    }

    return triangle;
}
```

Perustuu lähteeseen [3].

```
public static bool Intersects(List<Vertex> setA, List<Vertex> setB)
{
    Vector3 v = setA[0].Position - setB[0].Position;

    if (v.LengthSquared() == 0)
        return true;

    List<Vector3> simplex = new List<Vector3>();
    simplex.Add(v);

    bool originInsideTetrahedron = false;
    Vector3 d = -v;

    while (originInsideTetrahedron == false)
    {
        Vector3 w = Support.FurthestPtAlongDir(setA, d) -
                            Support.FurthestPtAlongDir(setB, -d);

        if (w.LengthSquared() == 0)
            return true;

        if (Vector3.Dot(w, d) < 0)
            return false;

        simplex.Add(w);
        DoSimplex(simplex, ref originInsideTetrahedron, ref d);

    }
    return true;
}
```

```
private static void DoSimplex(List<Vector3> simplex,
       ref bool originInsideTetrahedron, ref Vector3 dir)
{
    // Onko kysessä jana
    if (simplex.Count == 2)
        HandleLine(simplex, ref dir);

    // Onko kyseessä kolmio
    else if (simplex.Count == 3)
        HandleTriangle(simplex, ref dir);

    // Onko kyseessä kartio
    else
        HandleTetrahedron(simplex, ref originInsideTetrahedron, ref dir);
}
```

```
private static void HandleLine(List<Vector3> simplex, ref Vector3 d)
{
    Vector3 a = simplex[1];
    Vector3 b = simplex[0];

    Vector3 ab = b - a;
    Vector3 ao = -a;

    d = Vector3.Cross(Vector3.Cross(ab, ao), ab);
}
```

```
private static void HandleTriangle(List<Vector3> simplex, ref Vector3 dir)
{
    Vector3 a = simplex[2];
    Vector3 b = simplex[1];
    Vector3 c = simplex[0];

    Vector3 ao = -a;
    Vector3 ab = b - a;
    Vector3 ac = c - a;
    Vector3 abc = Vector3.Cross(ab, ac);

    if (Vector3.Dot(Vector3.Cross(abc, ac), ao) > 0)
    {
        simplex.RemoveAt(1);
        dir = Vector3.Cross(Vector3.Cross(ac, ao), ac);
    }
    else if (Vector3.Dot(Vector3.Cross(ab, abc), ao) > 0)
    {
        simplex.RemoveAt(0);
        dir = Vector3.Cross(Vector3.Cross(ab, ao), ab);
    }
    else if (Vector3.Dot(abc, ao) > 0)
    {
        simplex.Clear();
        simplex.Add(b);
        simplex.Add(c);
        simplex.Add(a);
        dir = abc;
    }
    else
    {
        dir = -abc;
    }
}
```

```
private static void HandleTetrahedron(List<Vector3> simplex,
    ref bool originInsideTetrahedron, ref Vector3 dir)
{
    Vector3 a = simplex[3];
    Vector3 b = simplex[2];
    Vector3 c = simplex[1];
    Vector3 d = simplex[0];

    if (OriginSameDirAsNormal(b, a, c))
    {
        simplex.RemoveAt(0); // Poistetaan piste d
        HandleTriangle(simplex, ref dir);
    }

    else if (OriginSameDirAsNormal(c, a, d))
    {
        simplex.RemoveAt(2); // Poistetaan piste b
        HandleTriangle(simplex, ref dir);
    }

    else if (OriginSameDirAsNormal(d, a, b))
    {
        simplex.RemoveAt(1); // Poistetaan piste c
        HandleTriangle(simplex, ref dir);
    }
    else
    {
        originInsideTetrahedron = true;
    }
}
```

```
private static bool OriginSameDirAsNormal(Vector3 a, Vector3 b, Vector3 c)
{
    Vector3 ab = b - a;
    Vector3 ac = c - a;
    Vector3 abc = Vector3.Cross(ab, ac);
    Vector3 ao = -a;
    return Vector3.Dot(abc, ao) > 0;
}
```

```
private static bool Intersects(List<Vertex> setA, List<Vertex> setB, ref Vector3 s)
{
    Vector3 p = new Vector3();
    Vector3 q = new Vector3();
    Vector3 w = new Vector3();
    Vector3 r1 = new Vector3();
    Vector3 r0 = new Vector3();
    List<Vector3> setR = new List<Vector3>();

    // Varmistus vain, että ei jäädä ikuiseen silmukkaan
    int maxLoops = setA.Count * setB.Count;
    for (int i = 0; i < maxLoops; i++)
    {
        r0 = r1;
        p = Support.FurthestPtAlongDir(setA, s);
        q = Support.FurthestPtAlongDir(setB, -s);
        r1 = Vector3.Normalize(q - p);

        // s • p ≤ s • q
        if (Vector3.Dot(s, r1) >= 0)
            return false;

        // Jos p ja q toistuvat
        if (i > 1 && r0 == r1)
            return false;

        // Jos tietyn iterointi kierroksen jälkeen  p ja q toistuvat,
        // mutta eivät peräkkäin, vaihdetaan etsintäsuunta(s = w)
        if (setR.Count > LOOPS_UNTIL_S_IS_W && setR.Contains(r1))
            s = w;
        else
        {
            // Jos pisteet eivät ole samalla pallon puoliskolla,
            // palautetaan törmäys
            if (setR.Count > 2 && !PointsOnSameHemisphere(setR, ref w))
                return true;

            // Kaava 15
            s = s - 2 * Vector3.Dot(r1, s) * r1;
            setR.Add(r1);
        }
    }// for i

    return true;
}
```

Perustuu lähteeseen [6].

```
private static bool PointsOnSameHemisphere(List<Vector3> points, ref Vector3 w)
{
    if (points.Count == 3)
        w = Vector3.Normalize(points[0] + points[1]);

    Vector3 ri = points[points.Count - 1];

    if (Vector3.Dot(w, ri) >= 0)
        return true;

    Matrix m = GetZAxisTransformationMatrix(ri);
    Vector3 z = Vector3.Transform(ri, m);
    Vector3 ra = Vector3.Transform(points[0], m);
    Vector3 rb = Vector3.Transform(points[1], m);

    if (ra.X * rb.Y - ra.Y * rb.X > 0)
    {
        Vector3 temp = ra;
        ra = rb;
        rb = temp;
    }

    for (int j = 2; j < points.Count - 2; j++)
     {
        Vector3 rj = Vector3.Transform(points[j], m);
        float rarj = ra.X * rj.Y - ra.Y * rj.X;
        float rbrj = rb.X * rj.Y - rb.Y * rj.X;

        if (rarj < 0 && rbrj > 0)
            continue;

        else if (rarj > 0 && rbrj > 0)
            ra = rj;

        else if (rarj < 0 && rbrj < 0)
            rb = rj;

        else
            return false;
    }

    Matrix mInv = Matrix.Invert(m);
    w = Vector3.Transform(Vector3.Normalize(ra + rb), mInv);

    return true;
}
```

Perustuu lähteeseen [5].

Z-akselin muunnos matriisi ja etäisin piste suunnasta

```
private static Matrix GetZAxisTransformationMatrix(Vector3 v)
{
    float oneOverLength = 1f / v.Length();

    Matrix m = new Matrix();
    m.M11 = v.X * v.Z * oneOverLength;
    m.M21 = v.Y * v.Z * oneOverLength;
    m.M31 = -(v.X * v.X + v.Y * v.Y) * oneOverLength;

    m.M12 = -v.Y * oneOverLength;
    m.M22 = v.X * oneOverLength;

    m.M13 = v.X;
    m.M23 = v.Y;
    m.M33 = v.Z;

    m.M44 = 1;
    return m;
}
```

```
class Support
{

    public static Vector3 FurthestPtAlongDir(List<Vertex> set, Vector3 d)
    {
        float maxDot = Vector3.Dot(d, set[0].Position);
        int index = 0;

        for (int i = 1; i < set.Count; i++)
        {
            float temp = Vector3.Dot(set[i].Position, d);
            if (temp > maxDot)
            {
                maxDot = temp;
                index = i;
            }
        }

        return set[index].Position;
    }
}
```

GJK ja säde pääalgoritmi.

```
public static bool Intersects(List<Vertex> setA, Vector3 vA,
    List<Vertex> setB, Vector3 vB)
{
    Vector3 r = vB - vA;
    Vector3 x = Vector3.Zero;
    Vector3 d = x - (setA[0].Position - setB[0].Position);
    List<Vector3> simplex = new List<Vector3>();
    Vector3 p = new Vector3();

    while (!CloseEnough(p, simplex, x)
)
    {
        Vector3 w = Support.FurthestPtAlongDir(setA, d)
                            - Support.FurthestPtAlongDir(setB, -d);

        // Jos d:n pistetulo (x - w):n kanssa on suurempi kuin nolla tarkoittaa se,
        // että x:n ohi ei päästy, joten lasketaan x:lle uusi kohta
        if (Vector3.Dot(d, (x - w)) > 0)
        {
            // Jos d:n pistetulo janan r kanssa on suurempi kuin nolla,
            // tarkoittaa se, että mikään piste janasta ei voi olla Minkowskin
            // erotuksen sisällä, koska jana mene poispäin, joten palautetaan
            // ei törmäystä
            if (Vector3.Dot(d, r) >= 0)
                return false;

            float lambda = Vector3.Dot(d, w) / Vector3.Dot(d, r);

            if(lambda > 1)
                return false;

            x = lambda * r;
            simplex.Clear();
        }

        simplex.Add(w);
        DoSimplex(simplex, ref p, x);
        d = x - p;
    }
    return true;
}
```

```
private static bool CloseEnough(Vector3 p, List<Vector3> simplex, Vector3 x)
{
    if (simplex.Count == 0)
        return false;

    float max = (x - simplex[0]).LengthSquared();

    for (int i = 1; i < simplex.Count; i++)
    {
        float temp = (x - simplex[i]).LengthSquared();

        if (temp > max)
            max = temp;
    }
    float px = (x - p).LengthSquared();
    return (px / max) < Epsilon;
}
```

LIITE 12

GJK ja säde pääalgoritmi.

```
private static void DoSimplex(List<Vector3> simplex, ref Vector3 p, Vector3 x)
{
    if (simplex.Count == 2)
        HandleLine(simplex, ref p, x);

    else if (simplex.Count == 3)
        HandleTriangle(simplex, ref p, x);

    else if (simplex.Count == 4)
        HandleTetrahedron(simplex, ref p, x);

    else
        p = simplex[0];
}
```

```
private static void HandleLine(List<Vector3> simplex, ref Vector3 p, Vector3 x)
{
    Vector3 a = simplex[1];
    Vector3 b = simplex[0];
    Vector3 ax = x - a;
    Vector3 ab = b - a;
    float t = Vector3.Dot(ax, ab) / Vector3.Dot(ab, ab);

    if (t > 1)
    {
        p = a;
        simplex.RemoveAt(0);
    }
    else
    {
        p = a + t * ab;
    }

}
```

```
private static void HandleTriangle(List<Vector3> simplex,
    ref Vector3 p, Vector3 x)
{
    Vector3 a = simplex[2];
    Vector3 b = simplex[1];
    Vector3 c = simplex[0];
    Vector3 ax = x - a;
    Vector3 ab = b - a;
    Vector3 ac = c - a;

    Vector3 abc = Vector3.Cross(ab, ac);

    if (Vector3.Dot(Vector3.Cross(abc, ac), ax) > 0)
    {
        simplex.RemoveAt(1);
        HandleLine(simplex, ref p, x);
    }
    else if (Vector3.Dot(Vector3.Cross(ab, abc), ax) > 0)
    {
        simplex.RemoveAt(0);
        HandleLine(simplex, ref p, x);
    }
    else if (Vector3.Dot(abc, ax) > 0)
    {
        abc.Normalize();
        p = PrimitiveTests.ClosestPtOnPlaneToPt(abc, a, x);
        simplex.Clear();
        simplex.Add(b);
        simplex.Add(c);
        simplex.Add(a);
    }
    else
    {
        abc.Normalize();
        p = PrimitiveTests.ClosestPtOnPlaneToPt(-abc, a, x);
    }
}
```

```
private static void HandleTetrahedron(List<Vector3> simplex,
       ref Vector3 p, Vector3 x)
{
    Vector3 a = simplex[3];
    Vector3 b = simplex[2];
    Vector3 c = simplex[1];
    Vector3 d = simplex[0];

    if (PointSameDirAsNormal(b, a, c, x))
    {
        simplex.RemoveAt(0); // Poistetaan piste d
        HandleTriangle(simplex, ref p, x);
    }

    else if (PointSameDirAsNormal(c, a, d, x))
    {
        simplex.RemoveAt(2); // Poistetaan piste b
        HandleTriangle(simplex, ref p, x);
    }

    else if (PointSameDirAsNormal(d, a, b, x))
    {
        simplex.RemoveAt(1); // Poistetaan piste c
        HandleTriangle(simplex, ref p, x);
    }
    else
    {
        p = x;
    }
}
```

Lähin piste janasta ja tasosta

```
public static Vector3 ClosestPtOnLineSegment(Vector3 a, Vector3 b, Vector3 p)
{
    Vector3 ab = b - a;

    float t = Vector3.Dot(p - a, ab) / Vector3.Dot(ab, ab);

    if (t <= 0.0f)
        return a;

    if (t >= 1.0f)
        return b;

    return a + t * ab;
}
```

```
public static Vector3 ClosestPtOnPlaneToPt(Vector3 planeNormal,
       Vector3 pointOnPlane, Vector3 point)
{
    float t = Vector3.Dot(planeNormal, point - pointOnPlane) /
             Vector3.Dot(planeNormal, planeNormal);

    return point - t * planeNormal;
}
```