Antti Partinen

YKSINKERTAISEN FYSIIKKAMOOTTORIN TOTEUTTAMINEN

Opinnäytetyö
Kajaanin ammattikorkeakoulu
Luonnontieteiden koulutusala
Tietojenkäsittelyn koulutusohjelma
29.11.2011

KAJAANIN AMMATTIKORKEAKOULU
UNIVERSITY OF APPLIED SCIENCES

OPINNÄYTETYÖ
TIIVISTELMÄ

Koulutusala Luonnontieteiden koulutusala	Koulutusohjelma Tietojenkäsittelyn koulutusohjelma
Tekijä(t) Antti Partinen	
Työn nimi Yksinkertaisen fysiikkamoottorin toteuttaminen	
Vaihtoehtoiset ammattiopinnot	Ohjaaja(t) Mikko Romppainen
	Toimeksiantaja mobilive Entertainment Ltd. Oy
Aika 29.11.2011	Sivumäärä ja liitteet 24 + 18

Pelikappaleiden fysiikka on nykyisin tärkeä osa pelien kehitystä. Pelimaailman esineiden halutaan reagoivan realistisesti pelaajaan toimintoihin, jotta tämä pystyy uppoutumaan pelimaailmaan. Fysiikka voi olla myös tärkeä osa pelimekaniikkaa. Varsinkin älypuhelimilla erilaiset fysiikkaan perustuvat pelit ovat nousseet massojen suosioon.

Työssä käydään aluksi läpi lyhyesti mitä fysiikalla tarkoitetaan peleissä ja miten fysiikkamoottorit ovat kehittyneet. Tämän jälkeen käsitellään tärkeitä käsitteitä fysiikkamoottorin ohjelmoinnin kannalta. Lopuksi esitellään työn aikana ohjelmoidun fysiikkamoottorin arkkitehtuuria ja toimintaa.

Työn tuloksena on yksinkertainen fysiikkamoottori, jonka käyttöönotto mobiilialustalla on pyritty tekemään mahdollisimman helpoksi. Tämän vuoksi moottori on ohjelmoitu C++-ohjelmointikielellä ja arkkitehtuurissa on pyritty ottamaan huomioon tarpeet jatkokehityksen kannalta. Fysiikkamoottori on ohjelmoitu peliä varten, jota kehittää mobilive Entertainment Ltd. Oy. Toteutettua fysiikkamoottoria voidaan pitää projektin kannalta onnistuneena, sillä vaikka siinä on huomattavia puutteita, soveltuu se peliin, jota varten se suunniteltiin.

Kieli	suomi
Asiasanat	ohjelmointi, pelit, fysiikkamoottori
Säilytyspaikka	☒ Verkkokirjasto Theseus ☒ Kajaanin ammattikorkeakoulun kirjasto

KAJAANIN AMMATTIKORKEAKOULU
UNIVERSITY OF APPLIED SCIENCES

THESIS
ABSTRACT

School Business	Degree Programme Business Information Technology
Author(s) Antti Partinen	
Title Implementing a Simple Physics Engine	
Optional Professional Studies	Instructor(s) Mikko Romppainen
	Commissioned by mobilive Entertainment Ltd. Oy
Date 29.11.2011	Total Number of Pages and Appendices 24 + 18

Physics simulation for game objects is nowadays an essential part of game development. It is important that the objects within the game world react realistically to player input so that the player can fully enjoy the experience. In some cases physics simulation can also be a crucial part of the game mechanics. Especially on smart phones different sorts of physics based games have gained popularity within the masses.

First, this thesis concentrates on what is meant by physics simulation in games and how physics engines have developed over time. Secondly, important concepts which are required for physics engine development are introduced.

The end result of this thesis is a simple physics engine which has been developed with ease of use on mobile platforms in mind. Due to this, the engine has been programmed with C++ programming language and the architecture has been designed so that further development is effortless. The physics engine has been programmed for a game which is being developed by mobilive Entertainment Ltd. Oy. The project can be considered successful, even though the created physics engine has many flaws as it can function in the game it was designed for.

Language of Thesis	Finnish
Keywords	programming, games, physics engine
Deposited at	☒ Electronic library Theseus ☒ Library of Kajaani University of Applied Sciences

SISÄLLYS

1 JOHDANTO

Objektien realistinen käyttäytyminen törmäyksissä on nykyisin tärkeä osa pelejä. Varsinkin älypuhelimilla erilaiset fysiikkaan perustuvat ongelmanratkontapelit ovat saavuttaneet suurta suosiota massojen keskuudessa. Tätä varten hyödynnetään peleissä usein erilaisia fysiikkamoottoreja. Vaikka markkinoilla on monia fysiikkamoottoreja, on näissä usein toimintoja, jotka saattavat vain hidastaa pelin kehitystä, mikäli niitä ei pelissä tarvita. Tämän takia yrityksen kannalta on helpompaa, mikäli heillä on käytössään oma fysiikkamoottori, jota on helppo muokata pelin tarpeita vastaaviksi.

Työssä esitellään tarvittavat käsitteet yksinkertaisen fysiikkamoottorin ohjelmointiin ja käydään läpi kaavat, joiden avulla kappaleiden liikkeet saadaan laskettua. Lisäksi käydään läpi miten fysiikkamoottorin tulisi toimia ja mitä on otettava huomioon fysiikkamoottoria suunniteltaessa.

Lopuksi käydään läpi työn tuloksena ohjelmoidun fysiikkamoottorin rakennetta ja ratkaisuja, joita on tehty fysiikkamoottoria ohjelmoidessa. Lisäksi esitellään tarkemmin tärkeät asiat fysiikkamoottorin toiminnan kannalta ja selitetään eri luokkien käyttötarkoitukset ja toiminta.

2 PELIEN FYSIIKKA

Fysiikka on valtava tiedonala, joka kuvaa eri kappaleiden tai hiukkasten käyttäytymistä. Vain erittäin pieni osa fysiikan eri osa-alueista on hyödyllistä peleissä. Vaikka esimerkiksi valon käyttäytymistä voitaisiin simuloida pelissä erilaisilla kaavoilla, ei pelien fysiikasta puhuttaessa usein tarkoiteta tätä. Vaikka tämäkin on fysiikkaa, pelien fysiikalla tarkoitetaan klassista mekaniikkaa eli sitä, miten erilaiset kappaleet käyttäytyvät törmätessään toisiinsa. (Millington, I. 2007, 2.)

Peleissä on käytetty fysiikkaa jo vuosikymmeniä. Aluksi pelien fysiikka oli yksinkertaista, mutta laskentatehon kasvaessa on peleissä mahdollista simuloida yhä monimutkaisempia fysikaalisia tilanteita. Nykyään peleistä löytyy jäykkien kappaleiden lisäksi pehmeitä kappaleita, kuten kankaita. (Millington, I. 2007, 2.)

Tärkeintä pelien fysiikassa on yrittää simuloida reaalimaailman tilanteita mahdollisimman tarkasti. Tämä antaa pelaajalle mahdollisuuden uppoutua peliin. Mikäli jokin kappale ei liiku vakuuttavasti, vetää tämä heti pelaajan ulos maailmasta. (Conger, D. 2004, 3.)

2.1 Fysiikkamoottorit

Vaikka fysiikkaa on käytetty peleissä vuosikymmeniä, ei fysiikkamoottori ole käsitteenä yhtä vanha. Alun perin, mikäli peliin tarvittiin fysiikkaa, ohjelmoitiin se aina erikseen pelin vaatimalla tavalla. Fysiikkamoottorit tulivat esille, kun haluttiin saada useita erilaisia, mutta samankaltaisia kappaleita toimimaan ilman, että niitä jokaista täytyi ohjelmoida erikseen. Fysiikkamoottori on siis käytännössä iso laskin, joka laskee miten kappale käyttäytyy sen perusteella, mitä arvoja sille annetaan. (Millington, I. 2007, 3.)

Fysiikkamoottorin käytöllä on kaksi selvää hyötyä. Toinen näistä on ajan säästö. Mikäli fysiikkaa halutaan käyttää useissa peleissä, erilaisissa tilanteissa toimivan fysiikkamoottorin tekeminen on nopeampaa kuin erilaisten tilanteiden ohjelmointi erikseen. Toinen hyöty on pelin fysiikan laatu. Mikäli peliä varten ohjelmoitaisiin omat fysiikka vettä, erilaisia kappaleita ja partikkeleja varten, olisi näiden yhdistäminen hankalaa. Mikäli yksi fysiikkamoottori pitää

huolen näistä kaikista, on todennäköisempää, että kaikki toimii niin kuin pitääkin. (Millington, I. 2007, 4.)

Fysiikkamoottorilla on myös omat haittansa. Luonnollisesti erilaisissa tilanteissa toimiva fysiikkamoottori on hidas, verrattuna yhtä tarkoitusta varten luotua ohjelmaa. Tämä on tärkeää huomioida varsinkin mobiililaitteilla. Lisäksi fysiikkamoottori vaatii aina suuren määrän erilaisia ominaisuuksia kappaleille ja ympäristöille, jotta se saa laskettua tarvittavat asiat. Tämän takia voi olla joskus yksinkertaisempaa luoda oma, tiettyyn tilanteeseen erikoistunut järjestelmä. (Millington, I. 2007, 4.)

3 PERUSKÄSITTEITÄ

Tässä luvussa käydään läpi peruskäsitteitä, joita tarvitaan fysiikkamoottorin ohjelmointiin.

3.1 Liikkeen lait

Fysiikkamoottorit perustuvat Newtonin kolmeen lakiin. Nämä lait määrittelevät tarkasti sen, miten pistemäinen massa käyttäytyy. Mikäli jäykän kappaleen keskipiste on sen massakeskipisteessä, käyttäytyy se lineaarisen liikkeen suhteen samalla tavalla kuin pistemäinen massa jolloin Newtonin lakeja voidaan hyödyntää. (Millington, I. 2007, 43.)

Newtonin ensimmäinen lain mukaan kappale jatkaa tasaista liikettä tai pysyy levossa, mikäli siihen vaikuttavien voimien summa on nolla. Newtonin toinen laki kertoo, että kappaleen kiihtyvyys on suoraan verrannollinen kappaleeseen vaikuttavaan voimaan ja kääntäen verrannollinen kappaleen massaan. Tästä saadaan kaava 1, joka on tärkeä kappaleen sijainnin laskemisen kannalta. Newton kolmannen lain mukaan voimalla on aina yhtä suuri vastavoima, mikä on tärkeää, kun myöhemmin käsitellään kappaleiden törmäyksiä. (Eberly, D. 2004, 30.)

$$a = \frac{F}{m} \quad (1)$$

3.2 Vektorit

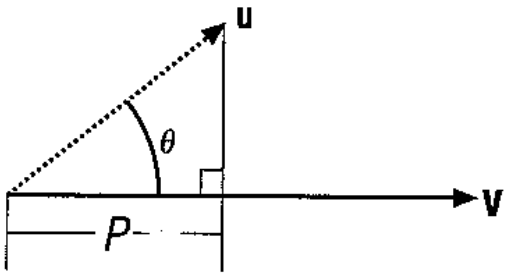
Vektori on suure, jolla on sekä suuruus että suunta. Tämän ansioista vektoreilla voidaan kuvata monia fysiikassa tarvittavia määreitä, kuten voimaa, nopeutta tai kiihtyvyyttä. Kaksiulotteista vektoria kuvataan usein kahdella luvulla; x ja y. Kolmiulotteisessa vektorissa on myös kolmas luku; z. Vektorin pituutta ei tallenneta erikseen vaan se voidaan laskea kaavalla 2. Kaksiulotteisen vektorin pituus lasketaan samalla kaavalla ilman z-arvoa. (Bourg, D. 2002, 286.)

$$|v| = \sqrt{x^2 + y^2 + z^2} \qquad (2)$$

Tärkeitä laskutoimituksia fysiikassa vektoreille ovat pistetulo ja ristitulo, joita tarvitaan usein kappaleiden törmäystarkistuksessa. Kahden kolmiulotteisen vektorin pistetulo lasketaan kaavan 3 mukaisesti.

$$P = u \cdot v = u_x * v_x + u_y * v_y + u_z * v_z \qquad (3)$$

Pistetulo kaksiulotteiselle vektorille lasketaan samalla tavalla ilman z-arvoja. Pistetulo (P) on skalaarisuure ja kuvaa tässä tapauksessa vektorin u heijastusta vektorille v, kuten kuviosta 1 käy ilmi. (Bourg, D. 2002, 292.)



Kuvio 1. Kahden vektorin pistetulo (Bourg, D. 2002, 293)

Ristitulolla saadaan laskettua kahta vektoria kohden kohtisuora vektori. Kahden kolmiulotteisen vektorin ristitulo lasketaan kaavan 4 mukaisesti.

$$u \times v = (u_y * v_z - u_z * v_y)i + (-u_x * v_z + u_z * v_x)j + (u_x * v_y - u_y * v_x)k \qquad (4)$$

Ristituloa voidaan käyttää apuna myös, mikäli halutaan selvittää ovatko vektorit yhdensuuntaiset, sillä yhdensuuntaisten vektorien ristitulo on aina nolla. Kaksiulotteiselle vektorille ei ristituloa voi määritellä samalla tavalla. Kaksiulotteiselle vektorille v saadaan sen sijaan vasemmalle ja oikealla kohtisuorat vektorit a ja b kaavoilla 5 ja 6. (Bourg, D. 2002, 291.)

$$a = (-v_y)i + (v_x)j \qquad (5)$$

$$b = (v_y)i + (-v_x)j \qquad (6)$$

Näiden lisäksi tärkeä operaatio vektoreille on normalisointi. Tämä tarkoittaa sitä, että muutetaan vektori yksikkövektoriksi. Käytännössä tämä tarkoittaa sitä, että otetaan vektori, joka on samansuuntainen alkuperäisen vektorin kanssa, mutta jonka pituus on yksi. Vektorista v saadaan yksikkövektori u kaavalla 7. Tämä toimii myös kaksiulotteiselle vektorille ilman z-arvoa. (Bourg, D. 2002, 287.)

$$u = \frac{v}{|v|} = v\left(\frac{v_x}{|v|}\right)i + \left(\frac{v_y}{|v|}\right)j + \left(\frac{v_z}{|v|}\right)k \quad (7)$$

4 JÄYKÄN KAPPALEEN FYSIIKKA

Jäykällä kappaleella tarkoitetaan kappaletta, joka voi kääntyä ja liikkua, mutta ei muuttaa muotoaan millään tavalla törmätessään toisiin kappaleisiin. Näin ollen jäykillä kappaleilla voidaan vain likimäärin simuloida todellisuutta. Tästä huolimatta on tämä tarpeeksi lähellä todellisuutta, jotta se näyttää uskottavalta. Varsinkin peleissä, joissa fysiikalta ei vaadita erityistä realistisuutta, ovat jäykät kappaleet riittäviä. (Stewart, 2011)

Käyttämällä jäykkiä kappaleita ovat tarvittavat laskut huomattavasti yksinkertaisempia. Pelien kannalta tämä on olennaista, sillä ne luonnollisesti vaativat, että kaikki suoritetaan reaaliajassa. (Stewart, 2011)

4.1 Kappaleen sijainti ja nopeus

Tässä kappaleessa käydään läpi erilaisia integraatiomenetelmiä, joilla jäykän kappaleen sijainti ja nopeus voidaan laskea. Samoja menetelmiä voidaan käyttää myös kappaleen kulman ja kulmanopeuden laskemiseen, kun tunnetaan kappaleen kulmakiihtyvyys.

Eulerin integraatio

Eulerin integraatio on yksinkertaisin tapa laskea kappaleen sijainti. Yksinkertaisuuden vuoksi Eulerin integraatiolla saadaan kappaleen sijainti tarkasti vain, jos kappaleen nopeus säilyy samana. Laskuvirhe saattaa olla pitemmällä aikavälillä huomattava, mutta mikäli pelin fysiikan ei tarvitse olla erityisen realistista, on Eulerin integraatio hyvä vaihtoehto sen yksinkertaisuuden vuoksi. Mikäli Eulerin integraatiota halutaan käyttää, mutta virheen tulisi olla mahdollisimman pieni, on mahdollista pienentää laskuvirhettä käyttämällä lyhyempää aikaväliä. (Gafferongames, 2011.)

Eulerin integraatiota käyttäen saadaan laskettua kappaleen sijainnin muutos Δp kaavan 8 mukaisesti kun tiedetään nopeus (v) ja aikaväli (Δt). Nopeuden muutos aikavälillä Δt saadaan selville kaavalla 9, kun tiedetään kappaleen kiihtyvyys (a). (Gafferongames, 2011.)

$$\Delta p = v * \Delta t \quad (8)$$

$$\Delta v = a * \Delta t \quad (9)$$

Runge-Kutta integraatio

Runge-Kutta integraatiosta käytetään yleisimmin neljännen kertaluvun integraatiota ja tämän takia tähän viitataankin yleensä RK4:nä. Kun on kyse pelien fysiikasta, Runge-Kutta integraatiossa virhe on tarpeeksi pieni, jotta sillä ei ole merkitystä. Runge-Kutta integraatiossa otetaan huomioon kiihtyvyyden muutos ajan kuluessa. Tämä tapahtuu arvioimalla kiihtyvyyttä ja nopeutta useissa eri kohdissa annetun aikavälin sisällä ja ottamalla näistä arvoista painotetun keskiarvon, jotta saadaan paras yksittäinen arvo tälle aikavälille. (Gafferongames, 2011.)

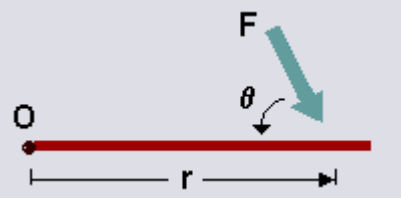
4.2 Voima ja kiihtyvyys

Voima on suure, jolla on suunta ja suuruus. Kun kappaleeseen vaikuttaa voima, aiheuttaa se kappaleelle kiihtyvyyden, joka on Newtonin toisen lain mukaisesti suoraan verrannollinen voiman suuruuteen ja käänteen verrannollinen kappaleen massaan. Kappaleen saavuttama kiihtyvyys voidaan laskea kaavan 1 mukaisesti kun tiedetään vaikuttava voima (F) ja kappaleen massa (m). (Eberly, D. 2004, 30.)

4.3 Vääntömomentti ja kulmanopeus

Vääntömomentti kertoo sen, miten paljon kappaleeseen vaikuttava voima aiheuttaa pyörimisliikettä. Vääntömomentti (T) saadaan laskettua kaavalla 10, kun tiedetään voiman vaikutuspisteen etäisyys (r) kappaleen massakeskipisteestä (O), voima joka kappaleeseen vaikuttaa (F) ja vaikuttavan voiman ja etäisyysvektorin välinen kulma (θ). Mikäli kappaleeseen vaikuttaa useita voimia, voidaan näiden vääntömomentit summata yhteen. (Physics, 2011.)

$$T = r * F * \sin(\theta) \quad (10)$$



Kuvio 2. Vääntömomentin laskemiseen tarvittavat suureet (Physics, 2011.)

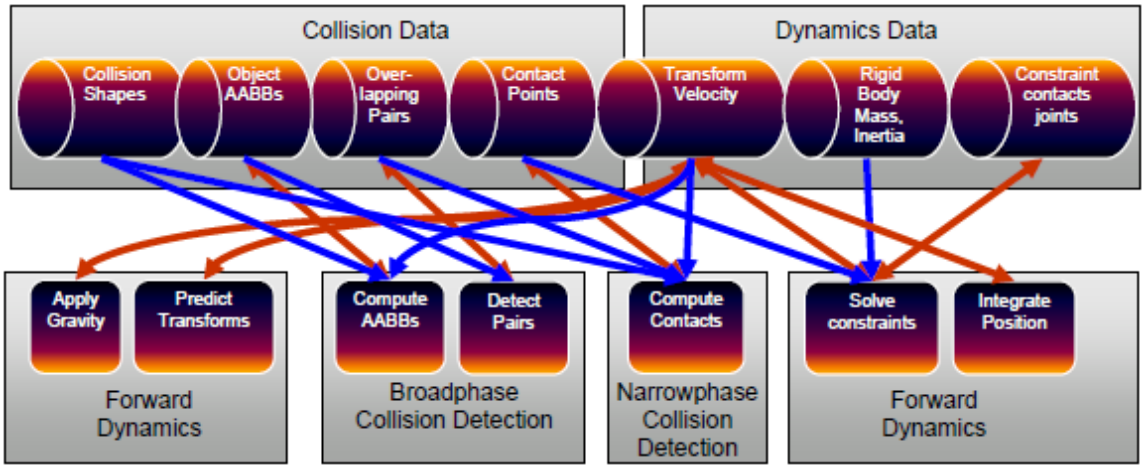
Kulmakiihtyvyyden laskukaava on hyvin samankaltainen kuin lineaarisen kiihtyvyyden. Eroina ovat massan tilalla hitausmomentti ja voiman tilalla vääntömomentti. (Millington, I. 2007, 198.)

Hitausmomentti kuvaa sitä, miten hankala kappaleen pyörimisnopeutta on muuttaa. Hitausmomentti riippuu siitä, mistä kappaletta pyöritetään. Yleisille kappaleille löytyy valmiit kaavat hitausmomentin laskemiseksi ja monimutkaiset kappaleet voidaan jakaa yksinkertaisiin osiin ja summata yhteen osien hitausmomentit. Kulmakiihtyvyys (α) saadaan laskettua kaavalla 11, kun tunnetaan vääntömomentti (τ) ja hitausmomentti (I). (Millington, I. 2007, 198.)

$$\alpha = \frac{\tau}{I} \quad (11)$$

5 TÖRMÄYSTEN KÄSITTELY

Tässä luvussa käsitellään sitä, millä tavalla fysiikkamoottori tunnistaa törmäykset sekä sitä, millä tavalla havaitut törmäykset käsitellään. Yksinkertaisimmillaan törmäysten käsittelyn voi sanoa koostuvan kahdesta vaiheesta; törmäyksien tunnistamisesta ja niihin reagoinnista. Kuvio 3 on esimerkki hyvin kattavasta Bullet-fysiikkamoottorin jäykän kappaleen fysiikan käsittelystä.



Kuvio 3. Bullet Physics Rigid Body Pipeline. (Coumans, E. 2010, 10)

5.1 Törmäystunnistus

Törmäystunnistus voi olla fysiikkamoottorin aikaa vievin osa. Pelin jokainen kappale voi mahdollisesti törmätä toiseen kappaleeseen, joten jokainen mahdollinen pari on tarkistettava. Mikäli pelissä on useita satoja monimutkaisia kappaleita, ei tämä ole mahdollista. Tämän takia järjestetään törmäystunnistus kahdessa vaiheessa. (Millington, I. 2007, 232.)

Törmäystunnistuksen ensimmäisessä vaiheessa kaikille kappaleille annetaan yksinkertainen muoto, kuten ympyrä tai AABB (axis-aligned bounding box). Tämän jälkeen suoritetaan törmäystarkistus kaikkien kappaleiden välillä käyttäen näitä muotoja. Tämän lisäksi voidaan käyttää erilaisia datarakenteita, joilla saadaan eliminoitua turhia tarkistuksia. On kuitenkin otettava huomioon, että tämän vaiheen on oltava erittäin nopea. On järkevämpää saada hie-

man useampia mahdollisia törmäyksiä kuin kuluttaa paljon aikaa muutaman törmäyksen eliminointiin. (Millington, I. 2007, 232.)

Rajausalueet ovat yksinkertaisia geometrisiä muotoja, joilla ympäröidään yksi tai useampi monimutkaisempi kappale. Yleisimpiä käytettyjä muotoja ovat ympyrät ja suorakulmiot. Rajausalueita käytetään, jotta voidaan suorittaa nopea törmäystarkistus ennen kuin siirrytään tarkempaan, aikaa vievään törmäystarkistukseen. (Ericson, C. 2005, 123.)

Kun harkitaan minkä muotoinen rajausalue kappaleelle annetaan, tulee ottaa huomioon, että mitä tiukemmin muoto vastaa kappaleen oikeaa muotoa, sitä vähemmän suoritetaan turhia törmäystarkistuksia. Usein tämä vaatii kuitenkin monimutkaisempaa rajausaluetta, joka vaatii enemmän muistia ja jonka törmäystarkistus vie enemmän aikaa. On siis tärkeää miettiä, että onko tarkempi törmäystarkistus niin vaativa, että monimutkaisemman rajausalueen törmäystarkistus säästää aikaa. On otettava huomioon, että rajausalueiden törmäystarkistus suoritetaan jokaisella päivityskerralla. Tämän lisäksi on otettava huomioon, että monimutkaisten rajausalueiden muodonkin laskeminen vaatii enemmän aikaa. Useimmiten rajausalue lasketaankin jo ennen ohjelman varsinaista suorittamista ja sitä muokataan tarpeen vaatiessa. (Ericson, C. 2005, 123.)

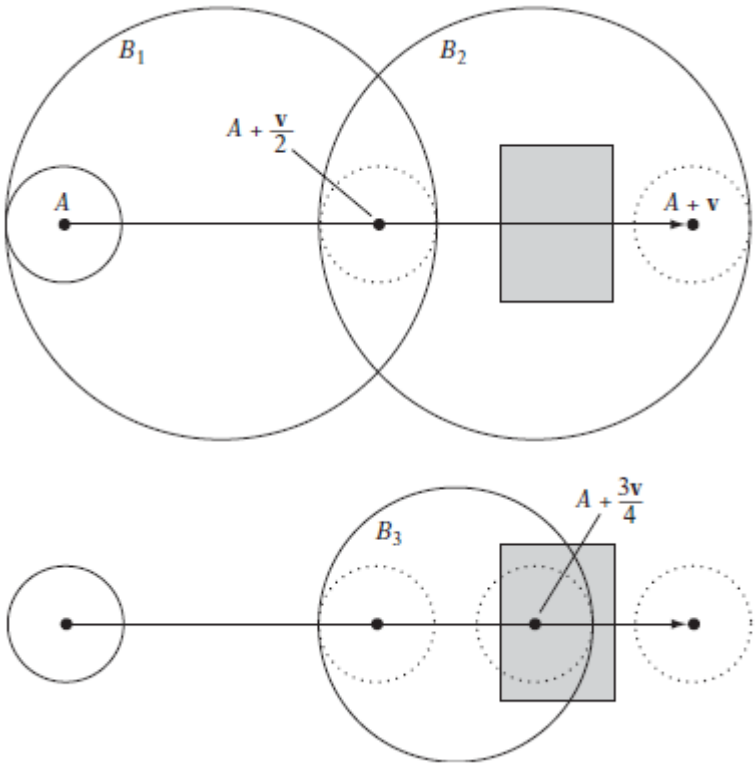
Toisessa vaiheessa suoritetaan tarkka törmäystunnistus, jossa otetaan huomioon kappaleiden muodot. Usein tässäkin vaiheessa yksinkertaistetaan muotoja jonkin verran, mikäli ne ovat erittäin monimutkaisia. Useimmiten yksinkertaisetkin muodot tarjoavat tarpeeksi realistisen törmäyksen ja ne ovat aina huomattavasti nopeampia laskea. Tätä vaihetta kutsutaan myös kontaktien luomiseksi, sillä tässä vaiheessa selvitetään törmäyksen tarkka sijainti ja kontaktin normaalivektori. Tämän lisäksi on hyvä myös selvittää, miten paljon kappaleet menevät sisäkkäin. (Millington, I. 2007, 232.)

5.1.1 Nopeiden kappaleiden törmäystunnistus

Nopeat kappaleet vaativat hieman ylimääräistä työtä, varsinkin, jos kyse on pienistä kappaleista. Mitä nopeammaksi kappaleen vauhti kasvaa, sitä todennäköisempää on, että kappale sijoitetaan kokonaan toisen kappaleen toiselle puolelle, jolloin törmäystä ei tunnisteta. (Ericson, C. 2005, 214.)

Yksi tapa ratkaista tämä ongelma olisi liikuttaa kappaletta pienin aikavälein niin, että se on aina osittain edellisen sijaintinsa päällä. Tämäkään ei kuitenkaan olisi riittävää tietynmuotoisille kappaleille, sillä kappaleiden väliin jää yhä aukkoja, joissa törmäystä ei tunnisteta. (Ericson, C. 2005, 214.)

Toinen ratkaisu käy ilmi kuviosta 4. Kappaletta liikutetaan ensin normaalisti, jonka jälkeen luodaan rajausalue, joka kattaa alueen, joka sisältää kappaleen alkuperäisen ja kappaleen uuden sijainnin. Tätä rajausaluetta testataan eri kappaleisiin ja mikäli se törmää johonkin kappaleeseen, johon liikutettu kappale itse ei törmää, asetetaan kappale sen alkuperäisen ja nykyisen sijainnin puoliväliin. Tämän jälkeen tarkistetaan törmäykset kappaleelle sekä rajausalueelle, jotka luodaan taas kappaleen molemmin puolin niin, että ne peittävät kappaleen eri sijainnit. Tätä toistetaan, kunnes löydetään sijainti, jossa kappaleet törmäävät. (Ericson, C. 2005, 214.)



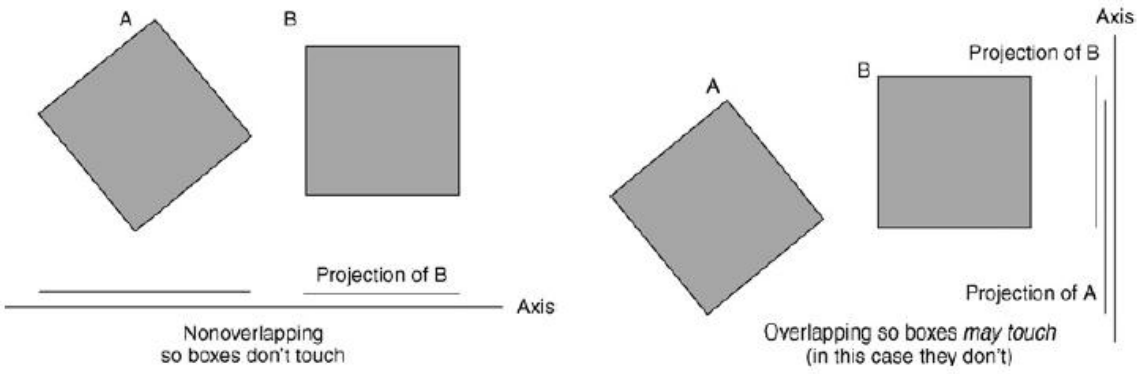
Kuvio 4. Tapa tunnistaa nopeiden kappaleiden törmäykset

Tätä tapaa käyttämällä on kuitenkin otettava huomioon, että tällä tavalla saatetaan löytää törmäys, vaikka kappaleet eivät koskaan törmäisikään toisiinsa. Näin voi käydä esimerkiksi

mikäli kappaleet liikkuvat yhdensuuntaisesti. Tämän takia onkin hyvä lopettaa törmäyksen etsiminen, mikäli se vaatii liikaa aikaa. (Ericson, C. 2005, 214.)

5.1.2 Separating Axis Theorem

Separating Axis Theorem (SAT) on yksi hyödyllisimmistä työkaluista törmäystunnistuksessa. SAT:n keskeinen ajatus on yksinkertainen; mikäli on olemassa sellainen suora, jolle heijastettuna kaksi kappaletta ei leikkaa toisiaan, kappaleet eivät ole kosketuksissa. Jotta SAT toimisi, kappaleiden tulee olla kuperia. Kappale on kupera, mikäli sen läpi ei ole mahdollista vetää viivaa, joka leikkaa kappaleen reunat useammassa kuin yhdessä kohtaa. (Eberly, D. 2004, 284.)



Kuvio 5. Kuperien kappaleiden törmäystarkistus (Millington, I. 2007, 290)

SAT:tä käytettäessä heijastetaan kappaleen jokainen nurkkapiste jokaiselle kappaleiden reunoja vastaaville akseleille. Heijastaminen tapahtuu ottamalla akselin ja pisteen pistetulo. Mikäli kappaleet eivät leikkaa jollakin näistä akseleista, eivät kappaleet ole päällekkäin. Muita akseleita ei tämän jälkeen enää tarvitse tarkistaa. (Eberly, D. 2004, 284.)

Kappaleiden kontaktinormaali saadaan laskettua etsimällä akseli, jolla kappaleet leikkaavat vähiten. Laskemalla miten paljon kappaleet leikkaavat tällä akselilla saadaan myös laskettua kuinka paljon kappaleet ovat uponneet toisiinsa. (Eberly, D. 2004, 284.)

5.2 Sisäkkäisten kappaleiden käsittely

Koska törmäystarkistusta ei pystytä suorittamaan jatkuvasti, ovat kappaleet yleensä liikkuneet jonkin verran sisäkkäin, kun törmäys huomataan. Tämän takia tulee kappaleet työntää erilleen, kun törmäys on tunnistettu. Tämän voi tehdä usealla eri tavalla, joista seuraavaksi esitellään kaksi. (Millington, I. 2007, 321.)

Yksinkertaisin tapa on liikuttaa objektia lineaarisesti kontaktin normaalivektorin suuntaisesti, kunnes kappale ei enää ole kosketuksissa toisen kappaleen kanssa. Kappaleita liikutettaessa tulisi ottaa huomioon kappaleiden massat niin, että painavampi kappale liikkuu vähemmän kuin kevyempi kappale. Tämän tavan ongelmana on sen realistisuuden puute muille kuin pyöreille kappaleille. (Millington, I. 2007, 321.)

Toinen, realistisempi tapa on muuten samanlainen, mutta siinä otetaan huomioon myös pyörimisliike. Kappaletta liikutetaan ja pyöritetään sen verran, että se ei enää koske toiseen kappaleeseen. Kuten aikaisemmin, lineaarinen liike on taas suhteellinen massaan verrattuna. Pyörimisliike sen sijaan on suhteellinen kappaleen hitausmomenttiin. Kappale, jolla on korkea hitausmomentti liikkuu siis lineaarisemmin kuin kappale, jolla on matala hitausmomentti. (Millington, I. 2007, 321.)

5.3 Kitkattomien törmäysten käsittely

Törmäykseen reagointi on seuraava vaihe törmäystunnistuksen jälkeen. Tässä vaiheessa käydään läpi jokainen löydetty törmäys ja lasketaan uudet nopeudet törmäykseen osallistuville kappaleille. (Hecker, 2011.)

Normaalisti törmäyksessä kappaleet vaikuttaisivat toisiinsa tietyllä voimalla lyhyen ajan, jonka jälkeen ne taas irtoaisivat toisistaan. Tämä ei kuitenkaan toimi jäykillä kappaleilla, sillä voima vaatisi aikaa suunnan muuttamiseen. Tässä tapauksessa objektit on saatava vaihtamaan suuntansa välittömästi, jotta ne eivät menisi toistensa sisään. Tämän takia otetaan käyttöön suure nimeltään impulssi. Impulssien avulla nopeuksia voidaan muuttaa välittömästi ja sitä voidaan ajatella suurena voimana erittäin lyhyellä aikavälillä. Uudet nopeudet kappaleille saadaan siis laskettua laskemalla impulssin suuruus törmäyshetkellä. (Hecker, 2011.)

Liikemäärä liittyy olennaisesti impulsseihin, sillä impulssi on yhtä suuri kuin liikemäärän muutos. Newtonin kolmannesta, eli voiman ja vastavoiman laista seuraa, että kun kaksi kappaletta törmää, niiden liikemäärä säilyy. Kappaleen liikemäärä on kappaleen massan ja nopeuden tulo. Impulssien laskemisen kannalta on olennaista, että oletetaan muiden kappaleeseen vaikuttavien voimien, kuten painovoiman, olevan mitättömiä. Useimmiten tämä pitääkin paikkaansa, sillä impulssi vaikuttaa erittäin lyhyellä aikavälillä. (Bourg, D. 2002, 89.)

Kuten jäykän kappaleen määritelmässä todettiin, jäykkä kappale ei muuta muotoaan törmäyksessä. Todellisuudessa kappale kuitenkin menettää aina osan energiastaan sen kuluessa kappaleen muodon muuttamiseen. Tästä syystä kappaleet eivät todellisuudessa jatka kimpoamista loputtomasti vaan ne menettävät liike-energiaansa kunnes pysähtyvät. Tämä on ominaisuus, jota fysiikkamoottorissakin tulee simuloida. Tätä varten on eri kappaleille määritelty kimmokertoimia, joiden avulla voidaan laskea kuinka suuri impulssi kappaleisiin todellisuudessa vaikuttaisi. Kimmokerroin on muuttuja, jota ei voida laskea kappaleen ominaisuuksista, vaan vaatii testausta. (Bourg, D. 2002, 89.)

$$j = \frac{(-1+e)*v^{AB}\cdot n}{n\cdot n\left(\frac{1}{M^A}+\frac{1}{M^B}\right)+\frac{\left(r_\perp^{AP}\cdot n\right)^2}{I^A}+\frac{\left(r_\perp^{BP}\cdot n\right)^2}{I^B}} \qquad (12)$$

Impulssi, joka ottaa huomioon sekä kulmanopeuden, että lineaarisen nopeuden, voidaan laskea kaavalla 12, kun tunnetaan kimmokerroin (e), kappaleiden välinen nopeus (v^{AB}), kontaktinormaali (n), kappaleiden massat (M), kappaleiden hitausmomentit (I) ja voiman vaikutuspisteen ja kappaleiden massakeskipisteiden välisten vektoreiden normaalivektorit ($r_\perp$). Törmäyksen jälkeiset nopeudet ja kulmanopeudet saadaan kaavoilla 13 ja 14. Törmäyksen toiseen kappaleeseen vaikuttaa samansuuruinen negatiivinen impulssi. (Hecker, 2011.)

$$v_2 = v_1 + \frac{j}{M}n \qquad (13)$$

$$\omega_2 = \omega_1 + \frac{r_\perp^{AP}\cdot jn}{I} \qquad (14)$$

6 OHJELMOINTIRAJAPINNAN SUUNNITTELU

API (Application programming interface) eli ohjelmointirajapinta antaa ohjelmoijalle mahdollisuuden kommunikoida helposti ohjelman kanssa. API antaa siis ohjelmojalle yksinkertaiset metodit, joilla kertoa ohjelmalle mitä sen tulisi tehdä. Näin ollen ohjelmoijan ei tarvitse ymmärtää itse ohjelman toimintaa vaan hän voi käyttää sitä API:n avulla. (PCMag, 2011.)

Toimivan API:n luominen on tärkeää, sillä se on usein juuri se osa ohjelmasta, jonka muut ohjelmoijat näkevät. Mikäli API ei toimi, saattavat kuluttajat etsiä ohjelman, jossa on parempi API tai vaatia jatkuvasti tukea API:n ymmärtämiseen. Lisäksi, mikäli API on julkinen, on sen muokkaaminen jälkikäteen hankalaa, sillä sen muokkaaminen rikkoisi kaikki aikaisemmat ohjelmat, jotka sitä käyttävät. Hyvän API:n tuntomerkkejä ovat muun muassa helppokäyttöisyys ja se, että sitä on hankala käyttää väärin. (APIDesign, 2011.)

API:n suunnitelu tulisi aloittaa pitämällä API mahdollisimman pienenä ja pyytämällä useita mielipiteitä. Suunniteltaessa API:a on pidettävä mielessä, että sen toiminnallisuuden pitäisi olla helposti selitettävissä. Mikäli jokin funktio on hankala nimetä, ei sitä usein ole järkevää lisätä. (APIDesign, 2011.)

7 TOTEUTUS

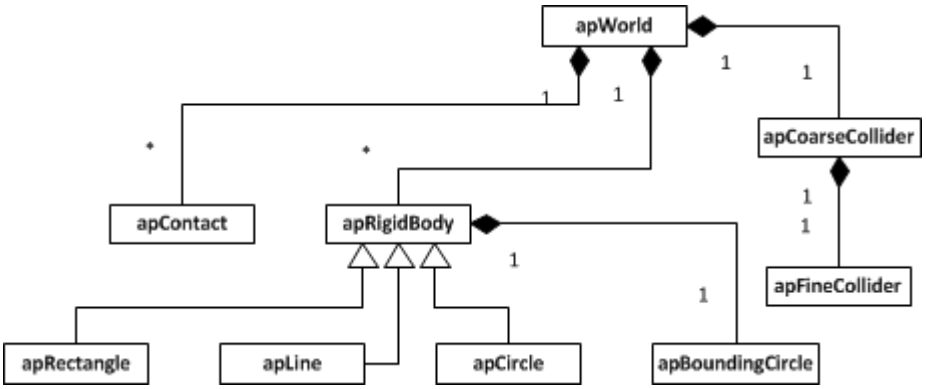
Opinnäytetyön aikana ohjelmoitiin yksinkertainen fysiikkamoottori. Fysiikkamoottori on rajattu kaksiulotteisiin kappaleisiin, sillä kolmiulotteisten kappaleiden fysiikka ei ole pelin kannalta tarpeellista. Lisäksi fysiikkamoottorissa on tuki vain nelikulmioille, ympyröille ja tasoille. Vaikka projekti on rajattu ulottumaan vain näihin kappaleisiin, on moottori suunniteltu niin, että erilaisten kappaleiden lisääminen on mahdollista. Fysiikkamoottori on ohjelmoitu C++-ohjelmointikielellä, jotta sitä voi käyttää suoraan mahdollisimman monella eri mobiilialustalla.

7.1 Vaatimukset

Fysiikkamoottori toteutetaan yksinkertaista kaksiulotteista peliä varten, joten sen vaatimukset ovat melko yksinkertaiset. Fysiikkamoottorilta vaaditaan seuraavan listan mukaisia asioita.

1. Tuki staattisille ja dynaamisille ympyröille ja nelikulmioille
2. Tuki staattisille tasoille
3. Kitkan aiheuttama pyörimisliike ympyrän ja toisen kappaleen välillä
4. Uusien kappaleiden lisääminen simulaatioon ajon aikana
5. Muutamien dynaamisten kappaleiden samanaikainen simulaatio

7.2 Fysiikkamoottorin arkkitehtuuri



Kuvio 6. Fysiikkamoottorin luokkakaavio

Työn tuloksena ohjelmoitu fysiikkamoottori on rakenteeltaan melko yksinkertainen. Kaiken perustana toimii apWorld-luokka, joka sisältää listat osoittimista kaikkiin kyseisen simulaation kappaleisiin sekä törmäyksiin, jotka ovat tapahtuneet yhden päivityksen aikana. Tämän lisäksi luokka sisältää suppeaan törmäystarkistukseen käytetyn apCoarseCollider-olion. Tämä olio puolestaan sisältää tarkkaan törmäystarkistukseen käytetyn apFineCollider-olion.

Simulaation kappaleet voivat olla kolmea eri tyyppiä, jotka on peritty apRigidBody-luokasta. Nämä eri tyypit ovat apRectangle, apLine ja apCircle. Tämän lisäksi apRigidBody-luokka sisältää apBoundingCircle-olion, jota käytetään suppeaan törmäystarkistukseen.

7.3 Luokkien kuvaus

Tässä kappaleessa käydään läpi fysiikkamoottorin luokat ja selitetään lyhyesti niiden tarkoitus ja toiminta.

apWorld

apWorld toimii fysiikkamoottorin perustana ja sillä on lista kaikista fysiikkamoottorin kappaleista. Tämä lista on tyyppiä std::vector ja sitä voidaan muokata vapaasti. Tämän lisäksi apWorld pitää yllä listaa jokaisen päivityskerran aikana tapahtuneista kontakteista. Myös kontakteja voidaan muokata vapaasti, mutta tätä ei suositella.

apWorldin stepSimulation-funktiota tulisi kutsua jokaisella ohjelman päivityskerralla halutulla aikavälillä. Tämä funktio siirtää kappaleita eteenpäin sekä laskee mahdolliset törmäykset ja niiden vaikutukset. Luokka sisältää myös maxIterations-muuttujan, jolla määritellään se, miten monta kertaa törmäystarkitus suoritetaan jokaisella päivityskerralla. Suurempi arvo vaatii enemmän suoritusaikaa, mutta estää tietyissä tapauksissa kappaleiden sisäkkäin menemisen.

apContact

apContact on yksinkertainen luokka törmäystietojen tallennusta varten. Luokka sisältää osoittimet törmäykseen osallistuviin kappaleisiin, törmäyksen sijainnin, törmäyksen normaalivektorin sekä sen, miten syvälle toisiinsa kappaleet ovat uponneet.

apCoarseCollider

apCoarseCollider suorittaa yksinkertaisen törmäystarkistuksen kahden kappaleen välillä. Mikäli luokka löytää mahdollisen törmäyksen, se muuttaa apRigidBody-olion oikeaksi aliluokaksi bodyType_-tunnisteen perusteella ja kutsuu apFineCollider-olion oikeaa törmäysfunktiota tarkemman törmäystarkistuksen suorittamiseksi. Tämän jälkeen luokka palauttaa apContact-olion, jonka sai palautusarvona apFineCollider-oliolta.

apFineCollider

apFineCollider sisältää tarvittavat funktiot erimuotoisten kappaleiden törmäystarkistusten suorittamiseksi. Luokka palauttaa apContact-luokan olion, joka on täytetty tarvittavilla tiedoilla tai 0, mikäli törmäystä ei tapahtunut.

apVector2

apVector2 on yksinkertainen kaksiulotteinen vektori, joka sisältää useita eri vektorilaskentaan käytettyjä funktioita sekä erilaisia apufunktiota vektorin käsittelyyn.

apRigidBody

apRigidBody sisältää kaikki tarvittavat tiedot kappaleen sijainnin laskemiseen. Näitä ovat käänteinen massa, kulma, sijainti, lineaarinen nopeus, kulmanopeus sekä vaikuttavat voimat. Näiden lisäksi apRigidBody-luokka sisältää apBoundingCircle-olion, jota käytetään suppeaan

törmäystarkistukseen. Kaikkia arvoja voidaan muokata vapaasti kulmaa ja käänteistä massaa lukuun ottamatta, joita muokatessa on käytettävä apufunktioita.

Kappaleen sijainti lasketaan uudestaan jokaisella päivityskerralla apWorldin kutsuessa sen stepSimulation-funktiota. Sijainti lasketaan Eulerin integraatiolla ottamalla huomioon kappaleen nopeus, kiihtyvyys sekä siihen vaikuttavat voimat. On suositeltavaa, että kappaletta liikutetaan kohdistamalla siihen erisuuruisia voimia eikä suoraan sen sijaintia muokkaamalla.

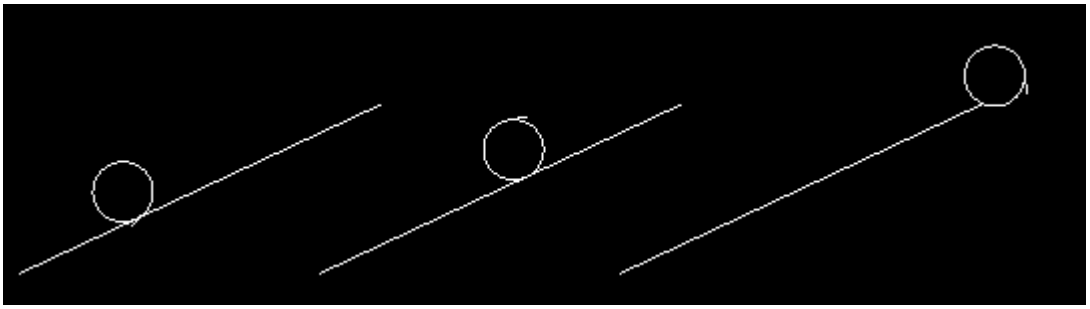
ApRigidBody toimii erimuotoisten kappaleiden yliluokkana. Näitä kappaleita ovat apCircle, apRectangle ja apLine. Nämä aliluokat eivät sisällä muuta kuin tarvittavat tiedot kappaleiden muodosta sekä funktion kappaleen hitausmomentin ja rajausalueen laskemiseksi.

7.4 Testaus

Projektin onnistumisen arvioimiseksi suoritettiin kolme testiä. Testit järjestettiin tietokoneella käyttäen OpenGL-grafiikkakirjastoa.

Testitapaus 1

Testataan kappaleiden välistä kitkaa asettamalla kolme eripainoista ympyrää kalteville tasoille. Annetaan ympyröille jatkuva vääntömomentti ja seurataan mitä tapahtuu.



Kuvio 7. Testijärjestely 1

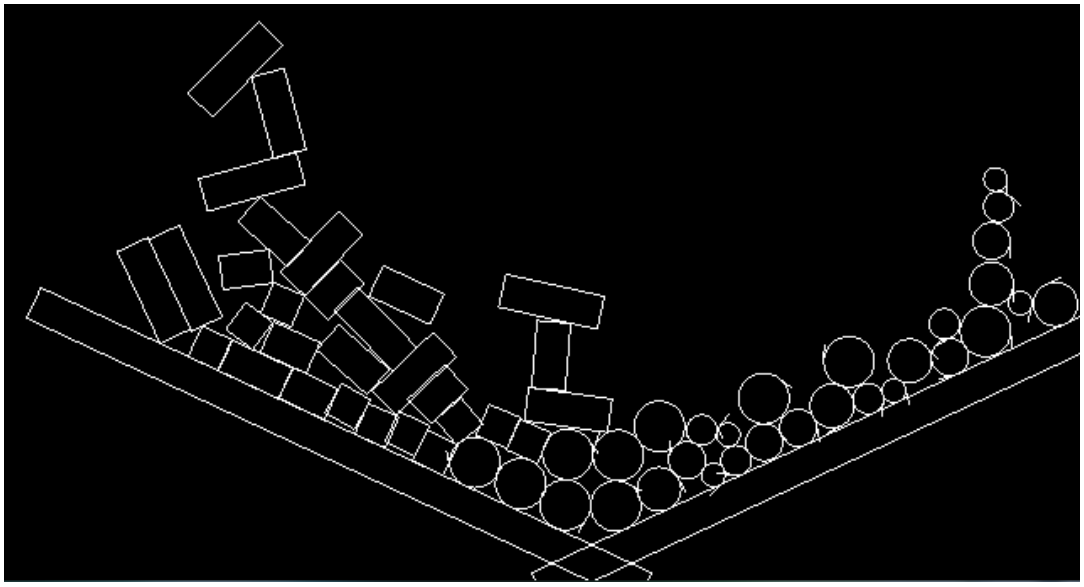
Kevyin pallo pyörii tasoa ylös, kun taas raskain pallo ei saa tarpeeksi pyörimisnopeutta vaan liukuu alas tasoa. Painoltaan näiden keskivälissä oleva pallo liukui hitaasti alaspäin, mutta lähti lopulta hitaasti ylöspäin. Tästä päätelleen kitka toimii kyseisillä kappaleilla, mutta vain osit-

tain. Kappaleet liukuvat enemmän kuin niiden pitäisi testijärjestelyssä käytetyllä kitkakertoimella.

Testitapaus 2

Testataan fysiikkamoottorin suorituskykyä lisäämällä kappaleita simulaatioon ja seurataan missä vaiheessa ruudunpäivitysnopeus tippuu liian alas. Simulaatioon lisätään sekä ympyröitä, että nelikulmioita ja maxIteration-muuttujan arvona on 3. Testi suoritetaan tietokoneella, joten sitä voidaan pitää vain suuntaa antavana mobiililaitteille.

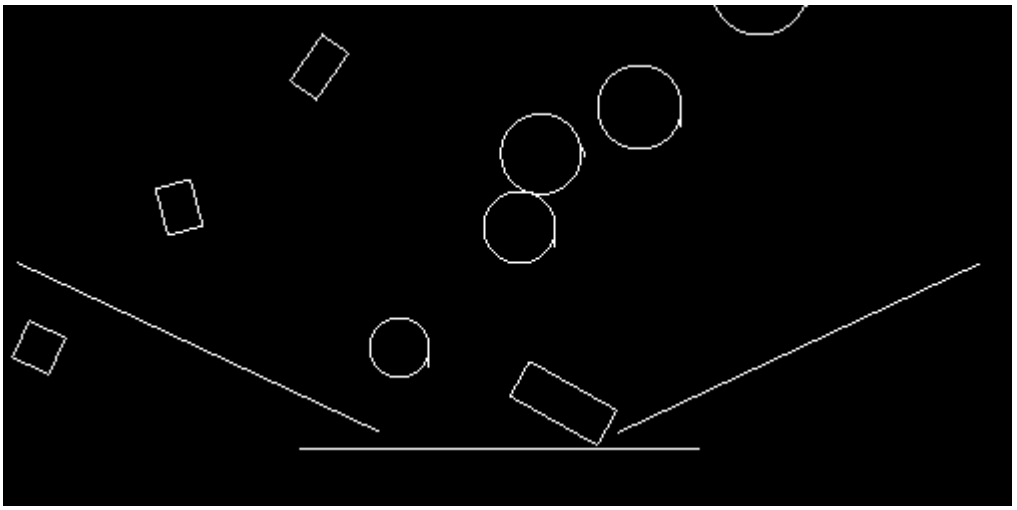
Kun simulaatioon oli lisätty noin 55 kappaletta, tippui ruudunpäivitysnopeus alle 60 ruutuun sekunnissa. Tästä voidaan päätellä, että suorituskyky on riittävä vähintään muutamalle kappaleelle mobiililaitteilla.



Kuvio 8. Testijärjestely 2

Testitapaus 3

Lisätään simulaatioon yksi dynaaminen neliö ja yksi dynaaminen ympyrä sekä useita staattisia ympyröitä ja neliöitä. Lisäksi asetetaan kaiken alle muutamia staattisia tasoja. Tarkastellaan dynaamisten kappaleiden reagointia staattisiin kappaleisiin.



Kuvio 9. Testijärjestely 3

Testin perusteella voidaan havaita, että kappaleet käyttäytyvät oletetulla tavalla kaikissa törmäyksissä, kun kimmokerroin on tarpeeksi suuri. Kun kimmokerroin asetetaan pieneksi, on neliön törmäyksiin reagointi epärealistista.

7.4.1 Vaatimusten ja tulosten vertailu

Kuviosta 9 käy ilmi millä testitapauksella on testattu mitäkin vaatimusta. Kuten kuviosta nähdään, täyttää fysiikkamoottori sille asetetut vaatimukset vain osittain. Vaikka kaikki vaatimukset täyttyvät, osa niistä ei täyty täydellisesti, sillä simulaatiossa on virheitä tietyissä tilanteissa. Jatkokehityksessä olisi hyvä keskittyä ensin näiden virheiden selvitykseen.

	Vaatimus 1	Vaatimus 2	Vaatimus 3	Vaatimus 4	Vaatimus 5
Testitapaus 1			(x)		
Testitapaus 2				x	x
Testitapaus 3	(x)	x			

Kuvio 10. Vaatimuksia vastaavat testitapaukset

8 YHTEENVETO

Projektia voidaan pitää tietyssä määrin onnistuneena, vaikka fysiikkamoottoriin jäi vielä selviä puutteita. Fysiikkamoottori vastaa puutteista huolimatta vaatimuksiin, jotka peli fysiikkamoottorille asettaa. Kappaleiden törmäystunnistus toimii ja kappaleet liikkuvat useimmiten odotetulla tavalla. Suurin ongelma jatkoa ajatellen on kappaleiden lepokontakti, sillä mikäli tällä hetkellä simulaatiossa on useita kappaleita päällekkäin, eivät ne asetu lepäämään toistensa päälle. Jatkossa olisi myös hyvä ottaa selvää, miksi kitka aiheuttaa tietyissä tapauksissa odottamattomia törmäyksiä kappaleiden välillä ja miksi kappaleet liukuvat enemmän kuin niiden pitäisi.

Fysiikkamoottoria olisi mahdollista optimoida huomattavasti sekä siihen olisi mahdollista lisätä erilaisia ominaisuuksia. Optimoinnin kannalta tärkeintä olisi muokata karkeaa törmäystarkistusta nopeammaksi, sekä muokata fysiikkamoottoria niin, että kappaleet menevät lepotilaan mikäli ne eivät liiku. Tämän lisäksi fysiikkamoottorin olisi hyvä lisätä tuki erilaisille nivelille. Optimoinnin kannalta tärkeää olisi myös luoda erillinen vaihtoehto staattisen kappaleiden lisäykseen. Tällä hetkellä staattisia kappaleita voi käyttää lisäämällä kappaleita, joiden käänteinen paino on nolla, jolloin ne eivät liiku minkään niihin törmätessä. Näille kappaleille suoritetaan kuitenkin turhia laskutoimituksia, joita staattiset kappaleet eivät tarvitsisi.

Etuna muihin fysiikkamoottoreihin verrattuina on fysiikkamoottorin yksinkertainen arkkitehtuuri. Tämän ansiosta on fysiikkamoottorin jatkokehittäminen ja muokkaaminen huomattavasti helpompaa kuin monien muiden fysiikkamoottoreiden.

Kun ottaa huomioon fysiikkamoottorin tämänhetkisen tilan, olisi suositeltavaa olla käyttämättä fysiikkamoottoria vielä tässä vaiheessa peliin, joka perustuu täysin fysiikkaan. Törmäyksiin reagointi ei ole tarpeeksi luotettavaa, jotta moottorin voitaisiin katsoa kelpaavaan kaupalliseen peliin. Tällä hetkellä fysiikkamoottori toimii silti hyvänä perustana jatkokehityksen kannalta ja sitä voi käyttää yksinkertaisissa peleissä, joissa otetaan erikseen huomioon fysiikkamoottorin puutteet.

LÄHTEET

APIDesign. http://lcsd05.cs.tamu.edu/slides/keynote.pdf Luettu 9.11.2011

Bourg, David. 2002. Physics for Game Developers

Erwin Coumans. 2010. Bullet 2.76 Physics SDK Manual

Conger, David. 2004. Physics Modeling for Game Programmers

Eberly, David. 2004. Game Physics

Ericson, Christer. 2005. Real-Time Collision Detection

Gafferongames. http://gafferongames.com/game-physics/integration-basics/ Luettu 15.11.2011

Hecker. http://chrishecker.com/Rigid_Body_Dynamics Luettu 28.10.2011

Millington, Ian. 2007. Game Physics Engine Development

PCMag. http://www.pcmag.com/encyclopedia_term/0,2542,t=application+programming+interface&i=37856,00.asp#fbid=0oFdutn8Ex4 Luettu 4.11.2011

Physics. http://www.physics.uoguelph.ca/tutorials/torque/Q.torque.intro.html Luettu 9.11.2011

Stewart. http://www.amath.washington.edu/courses/533-winter-2011/Stewart00.pdf Luettu 5.11.2011

LIITTEET

LIITE 1: LUOKKIEN DOKUMENTAATIO

CLASS DOCUMENTATION

apBoundingCircle Class Reference

`#include <apRigidBody.h>`

Public Member Functions

- **apBoundingCircle** (float circleRadius)

Public Attributes

- float **radius**

Detailed Description

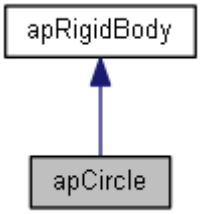
Simple, circular bounding area

The documentation for this class was generated from the following file:

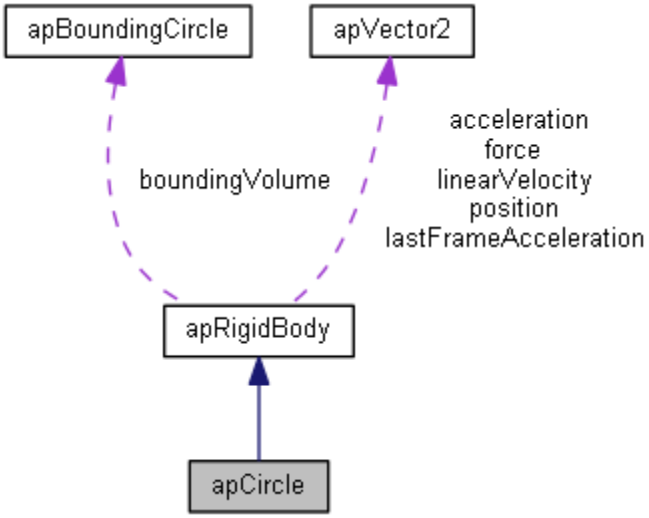
- C:/Users/Antti/Documents/Visual Studio 2010/Projects/apPhysics/apPhysics/apRigidBody.h

apCircle Class Reference

Inheritance diagram for apCircle:



Collaboration diagram for apCircle:



Public Member Functions

- float **getRadius** () const
- **apCircle** (float circleRadius)

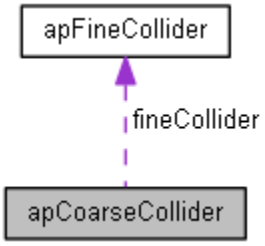
The documentation for this class was generated from the following files:

- C:/Users/Antti/Documents/Visual Studio 2010/Projects/apPhysics/apPhysics/apRigidBody.h
- C:/Users/Antti/Documents/Visual Studio 2010/Projects/apPhysics/apPhysics/apRigidBody.cpp

apCoarseCollider Class Reference

`#include <apCoarseCollider.h>`

Collaboration diagram for apCoarseCollider:



Public Member Functions

- **apContact * collideBodies** (**apRigidBody** *one, **apRigidBody** *two)

Public Attributes

- **apFineCollider fineCollider**

Detailed Description

Performs simple collision detection between the bounding areas of the rigid bodies and calls **apFineCollider** for every possible collision

---

Member Function Documentation

apContact * apCoarseCollider::collideBodies (apRigidBody * *one*, apRigidBody * *two*)

Checks if the bodies collide. Return 0 if there's no collision, if there is a collision, calls fineCollider for proper collision.

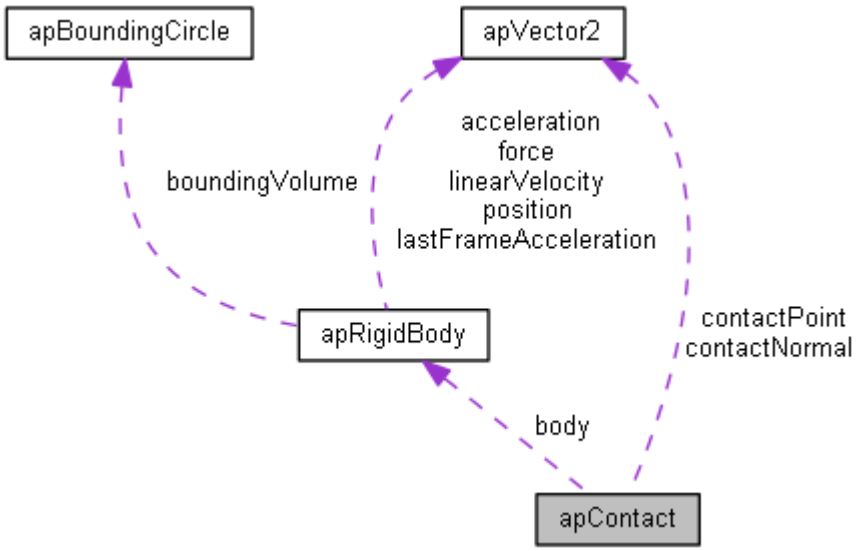
The documentation for this class was generated from the following files:

- C:/Users/Antti/Documents/Visual Studio 2010/Projects/apPhysics/apPhysics/apCoarseCollider.h
- C:/Users/Antti/Documents/Visual Studio 2010/Projects/apPhysics/apPhysics/apCoarseCollider.cpp

apContact Class Reference

`#include <apContact.h>`

Collaboration diagram for apContact:



Public Attributes

- **apRigidBody** * **body** [2]
- **apVector2 contactPoint**
- **apVector2 contactNormal**
- float **penetration**

Detailed Description

A simple storage class to hold the necessary contact data

Member Data Documentation

apRigidBody* apContact::body[2]

Holds the bodies which are in contact

apVector2 apContact::contactNormal

Direction of the contact in world coordinates

apVector2 apContact::contactPoint

Position of the contact in the world

float apContact::penetration

Penetration depth of the contact point

The documentation for this class was generated from the following file:

- C:/Users/Antti/Documents/Visual Studio 2010/Projects/apPhysics/apPhysics/apContact.h

apDebugDrawer Class Reference

```
#include <apDebugDrawer.h>
```

Detailed Description

A class to draw simple shapes which can be used to debug physics. Not yet done...

The documentation for this class was generated from the following file:

- C:/Users/Antti/Documents/Visual Studio 2010/Projects/apPhysics/apPhysics/apDebugDrawer.h

apFineCollider Class Reference

`#include <apFineCollider.h>`

Public Member Functions

- **apContact** * **circleAndCircle** (const **apCircle** *one, const **apCircle** *two)
- **apContact** * **circleAndLine** (const **apCircle** *circle, const **apLine** *line)
- **apContact** * **rectAndLine** (const **apRectangle** *rect, const **apLine** *line)
- **apContact** * **rectAndCircle** (const **apRectangle** *rect, const **apCircle** *circle)
- **apContact** * **rectAndRect** (const **apRectangle** *one, const **apRectangle** *two)

Detailed Description

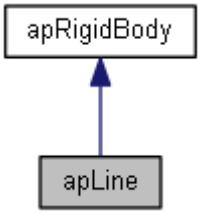
Takes care of checking collision properly

The documentation for this class was generated from the following files:

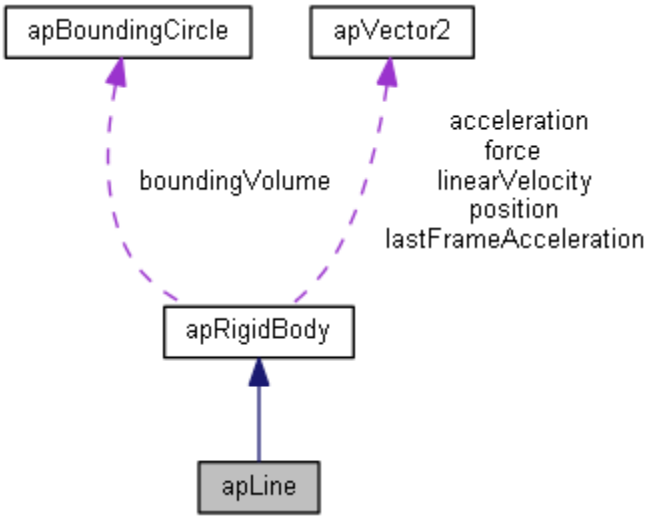
- C:/Users/Antti/Documents/Visual Studio 2010/Projects/apPhysics/apPhysics/apFineCollider.h
- C:/Users/Antti/Documents/Visual Studio 2010/Projects/apPhysics/apPhysics/apFineCollider.cpp

apLine Class Reference

Inheritance diagram for apLine:



Collaboration diagram for apLine:



Public Member Functions

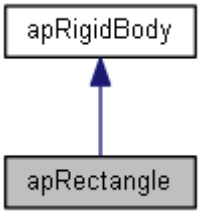
- float **getLength** () const
- **apLine** (float lineLength)

The documentation for this class was generated from the following files:

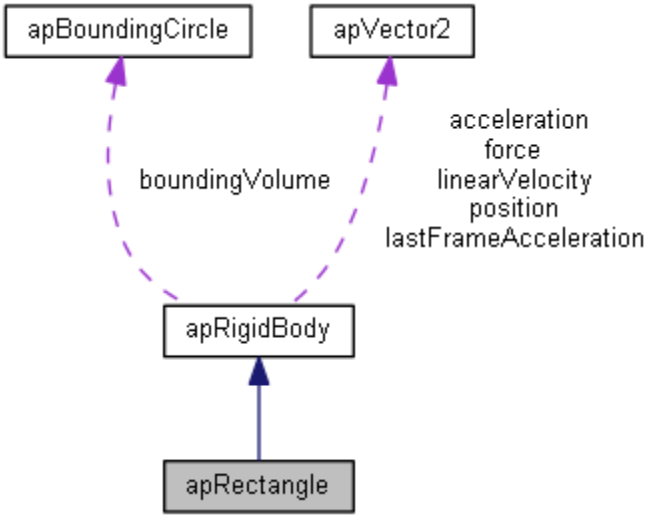
- C:/Users/Antti/Documents/Visual Studio 2010/Projects/apPhysics/apPhysics/apRigidBody.h
- C:/Users/Antti/Documents/Visual Studio 2010/Projects/apPhysics/apPhysics/apRigidBody.cpp

apRectangle Class Reference

Inheritance diagram for apRectangle:



Collaboration diagram for apRectangle:



Public Member Functions

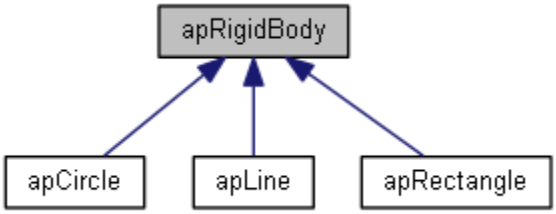
- **apRectangle** (float width, float height)
- float **getHalfWidth** () const
- float **getHalfHeight** () const

The documentation for this class was generated from the following files:

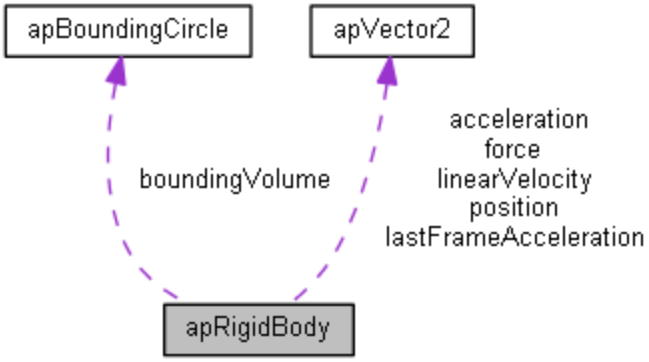
- C:/Users/Antti/Documents/Visual Studio 2010/Projects/apPhysics/apPhysics/apRigidBody.h
- C:/Users/Antti/Documents/Visual Studio 2010/Projects/apPhysics/apPhysics/apRigidBody.cpp

apRigidBody Class Reference

Inheritance diagram for apRigidBody:



Collaboration diagram for apRigidBody:



Public Member Functions

- void **setAngle** (const float angleRadians)
- void **setAngleInDegrees** (const float angleDegrees)
- float **getAngleInDegrees** () const
- float **getAngle** () const
- void **setMass** (const float mass)
- void **setInverseMass** (const float inverseMass)
- const float **getMass** ()
- const float **getInverseMass** ()
- const int **getBodyType** ()
- void **stepSimulation** (const float timeStepSeconds)
- void **addForce** (const **apVector2** &addedForce)
- void **addTorque** (const float addedTorque)
- void **addForceAt** (const **apVector2** &addedForce, const **apVector2** &point)
- void **addForceAtWorldPoint** (const **apVector2** &addedForce, const **apVector2** &point)
- void **clearForces** ()

Public Attributes

- **apVector2 position**
- **apVector2 linearVelocity**
- **apVector2 acceleration**
- **apVector2 lastFrameAcceleration**
- float **angularVelocity**
- float **linearDamping**
- float **angularDamping**
- **apBoundingCircle boundingVolume**
- **apVector2 force**

- float **torque**
- float **inertia**
- float **restitution**
- float **friction**

Protected Member Functions

- **apRigidBody** ()
- virtual void **recalculateInternal** ()

Protected Attributes

- float **inverseMass_**
- float **angle_**
- int **bodyType_**

Constructor & Destructor Documentation

apRigidBody::apRigidBody () `[protected]`

Protected, we don't want anyone to create just a rigid body

Member Function Documentation

void apRigidBody::addForce (const apVector2 & *addedForce*)

Adds a force which affects linear acceleration for the next step only

void apRigidBody::addForceAt (const apVector2 & *addedForce*, const apVector2 & *point*)

Adds a force to a specific point which affects acceleration for the next step only. Point is relative to the body and can be outside of the body but cannot be 0,0 (Use addForce if you add force directly to the center because it wouldn't cause any torque anyway)

void apRigidBody::addForceAtWorldPoint (const apVector2 & *addedForce*, const apVector2 & *point*)

Adds a force to a specific point which affects acceleration for the next step only. Point is in world coordinates

void apRigidBody::addTorque (const float *addedTorque*)

Adds a force which affects angular acceleration for the next step only

void apRigidBody::clearForces ()

Clears force and torque. This is called after each step

float apRigidBody::getAngle () const [inline]

Get the angle of the body in radians

float apRigidBody::getAngleInDegrees () const

Get the angle of the body in degrees

const int apRigidBody::getBodyType () [inline]

Returns the body type of the object

const float apRigidBody::getInverseMass () [inline]

Returns inverse mass of the object

const float apRigidBody::getMass ()

Returns the mass of the object

void apRigidBody::recalculateInternal () [protected, virtual]

Reimplement for all subclasses! Recalculate inertia and bounding volume.

void apRigidBody::setAngle (const float *angleRadians*)

Sets the angle of the body. Makes sure it stays between -PI and PI

void apRigidBody::setAngleInDegrees (const float *angleDegrees*)

Sets the angle of the body in degrees. Makes sure it stays between -PI and PI

void apRigidBody::setInverseMass (const float *inverseMass*)

Sets the inverse mass directly

void apRigidBody::setMass (const float *mass*)

Sets the mass of the object (takes care of inversing it)

void apRigidBody::stepSimulation (const float *timeStepSeconds*)

Steps the simulation forward by a specified time

Member Data Documentation

apVector2 apRigidBody::acceleration

Can be used to set a constant acceleration for the particle (for example, gravity)

float apRigidBody::angle_ [protected]

angle of the body in radians. Between (-pi, pi]. Use setAngle to change directly

float apRigidBody::angularDamping

Slows down angular motion, can be used to simulate various things. Tells us how much velocity the object retains after the update 1 = no damping, 0 = loses all velocity

float apRigidBody::angularVelocity

angular velocity of the body

int apRigidBody::bodyType_ [protected]

Type of the collision area

apBoundingCircle apRigidBody::boundingVolume

Bounding volume for the body, used for coarse collision check

apVector2 apRigidBody::force

Force applied at the current step

float apRigidBody::friction

How much friction there is between the colliding bodies. Should be set somewhere between 0 and 1. The friction value used in the collision is the average of the friction values of the colliding bodies. NOTE: All collision can be pretty weird on high friction values

float apRigidBody::inertia

Inertia of this rigid body

float apRigidBody::inverseMass_ [protected]

1 / mass. Having the inverse mass is more useful as we can't have object with 0 mass, but having objects with infinite mass is useful

apVector2 apRigidBody::lastFrameAcceleration

our linear acceleration during the previous frame.

float apRigidBody::linearDamping

Slows down linear motion, can be used to simulate various things. Tells us how much velocity the object retains after the update 1 = no damping, 0 = loses all velocity

apVector2 apRigidBody::linearVelocity

linear velocity of the body

apVector2 apRigidBody::position

linear position (world position) of the body

float apRigidBody::restitution

How much the objects bounces in collisions. Should be set somewhere between 0 and 1. The restition value used in the collision is the average of the restitution values of the colliding bodies. NOTE: Rectangle collision are rather weird on low restitution values

float apRigidBody::torque

Torque applied at the current step

The documentation for this class was generated from the following files:

- C:/Users/Antti/Documents/Visual Studio 2010/Projects/apPhysics/apPhysics/apRigidBody.h
- C:/Users/Antti/Documents/Visual Studio 2010/Projects/apPhysics/apPhysics/apRigidBody.cpp

apVector2 Class Reference

`#include <apVector2.h>`

Public Member Functions

- **apVector2** (const float x, const float y)
- void **invert** ()
- float **length** () const
- float **squareLength** () const
- void **normalize** ()
- void **addScaledVector** (const **apVector2** &vector, float scale)
- float **dotProduct** (const **apVector2** &vector) const
- **apVector2 getPerpVector** () const
- **apVector2 getInvert** () const
- float **perpDotProduct** (const **apVector2** &vector) const
- **apVector2 projectTo** (const **apVector2** &vector) const
- void **setZero** ()
- float **angleBetween** (const **apVector2** &vector) const
- float **distanceBetween** (const **apVector2** &vector) const
- void **operator*=** (const float value)
- **apVector2 operator*** (const float value) const
- void **operator+=** (const **apVector2** &v)
- **apVector2 operator+** (const **apVector2** &v) const
- void **operator-=** (const **apVector2** &v)
- **apVector2 operator-** (const **apVector2** &v) const

Public Attributes

- float **x**
- float **y**

Detailed Description

A simple 2d-vector with some helper functions

Member Function Documentation

void apVector2::addScaledVector (const apVector2 & *vector*, float *scale*) `[inline]`

Adds a vector which is scaled by a certain amount to this vector

float apVector2::angleBetween (const apVector2 & *vector*) const `[inline]`

Returns the angle between the vectors

float apVector2::distanceBetween (const apVector2 & *vector*) const `[inline]`

Return the distance between the vectors

float apVector2::dotProduct (const apVector2 & *vector*) const [inline]

Returns the dot product

apVector2 apVector2::getPerpVector () const [inline]

Return the (left?) perpendicular vector

void apVector2::invert () [inline]

Flips the components

float apVector2::length () const [inline]

Returns the length of this vector

void apVector2::normalize () [inline]

Normalizes a non-zero vector

apVector2 apVector2::operator* (const float *value*) const [inline]

Returns a copy of the the vector multiplied with the given value

void apVector2::operator*= (const float *value*) [inline]

Multiplies the vector with the given value

apVector2 apVector2::operator+ (const apVector2 & *v*) const [inline]

Returns a copy of the added vectors

void apVector2::operator+= (const apVector2 & *v*) [inline]

Adds the vector to this vector

apVector2 apVector2::operator- (const apVector2 & *v*) const [inline]

Returns a copy of the subtracted vectors

void apVector2::operator-= (const apVector2 & *v*) [inline]

Subtracts the vector from this vector

float apVector2::perpDotProduct (const apVector2 & *vector*) const [inline]

Returns the perp dot product

apVector2 apVector2::projectTo (const apVector2 & *vector*) const [inline]

Returns a vector which is a result of projecting this vector to an axis

void apVector2::setZero () [inline]

Sets both coordinates to zero

float apVector2::squareLength () const [inline]

Returns the squared length

The documentation for this class was generated from the following file:

- C:/Users/Antti/Documents/Visual Studio 2010/Projects/apPhysics/apPhysics/apVector2.h

apWorld Class Reference

`#include <apWorld.h>`

Public Member Functions

- void **stepSimulation** (float timeStepSeconds)

Public Attributes

- int **maxIterations**
- std::vector< **apContact** * > **contacts**
- std::vector< **apRigidBody** * > **bodies**

Detailed Description

Basis of the physics simulation.

Member Function Documentation

void apWorld::stepSimulation (float *timeStepSeconds*)

Steps the simulation forward by a given time step. This should be called in the game update loop

---

Member Data Documentation

std::vector<apRigidBody*> apWorld::bodies

List of all the bodies which take part in the physics simulation in this **apWorld**

std::vector<apContact*> apWorld::contacts

List of all the contacts found in the current step

int apWorld::maxIterations

The maximum amount of contact generation/resolving we'll perform in a single step. Increase this value if objects keep penetrating each other. Each iteration possibly doubles the time required for each step so don't increase this unless it's absolutely necessary.

The documentation for this class was generated from the following files:

- C:/Users/Antti/Documents/Visual Studio 2010/Projects/apPhysics/apPhysics/apWorld.h
- C:/Users/Antti/Documents/Visual Studio 2010/Projects/apPhysics/apPhysics/apWorld.cpp