Jyri Lehto

Fysiikkamoottorit pelikehityksessä

Tietotekniikan
pro gradu -tutkielma
25. syyskuuta 2012

Jyväskylän yliopisto

Tietotekniikan laitos

Jyväskylä

Tekijä: Jyri Lehto
Yhteystiedot: lehto.jyri@gmail.com
Työn nimi: Fysiikkamoottorit pelikehityksessä
Title in English: Physics Engines in Game Development
Työ: Tietotekniikan pro gradu -tutkielma
Sivumäärä: 106
Tiivistelmä: Tutkielmassa tarkastellaan fysiikkamoottoreiden toimintaa XNA-kehyksessä. Konstruktiivisessa osassa toteutetaan adapterikirjasto, joka mahdollistaa fysiikkamoottoreiden nopean ja sujuvan vaihdon ja testaamisen.
Abstract: This thesis describes the functionality of physics engines with XNA framework. At the constructive phase, adapter library component is being implemented to make changing and testing of different physics engines easier.
Avainsanat: fysiikka, fysiikkakirjasto, fysiikkamoottori, Jypeli, pelikehys, pelikirjasto, pelimoottori, tietokonepelit, XNA kehys
Keywords: computer games, game engine, game framework, game library, Jypeli, physics, physics engine, physics library, XNA framework

Copyright © 2012 Jyri Lehto

All rights reserved.

Esipuhe

Työn aihepiiri tuli tutuksi Jyväskylän yliopiston järjestämän nuorten peliohjelmointikurssin kautta. Ensimmäinen nuorten peliohjelmointikurssi järjestettiin kesällä 2009, ja siitä lähtien kursseja on järjestetty säännöllisesti joka kesä. Kurssien määrä kesää kohti riippuu osanottajien määrästä, ja määrä on ollut jatkuvassa vuosittaisessa kasvussa. Itse en varsinaisesti ole kyseisellä kurssilla ollut, mutta luentoja sivusta seuranneena voin todeta, että nuorten peliohjelmointikurssilla on vielä paljon annettavaa nykynuorisolle. Paljon se on jo antanutkin.

Kiitän kaikkia tutkielman valmiiksi saattamiseen myötävaikuttaneita henkilöitä: ohjaaja lehtori Vesa Lappalainen, apuohjaajat Tero Jäntti, Tomi Karppinen ja tohtorikoulutettava Antti-Jussi Lakanen.

Sanasto

Adapteri käsitetään yleisemmin joksikin laitteeksi, joka esimerkiksi sähkötekniikassa mahdollistaa erimuotoisten liitinten yhteensovittamisen. Tässä tutkielmassa pyritään kehittämään kirjasto, joka toimii ideologisesti adapterin tavoin.

Aktuaattori tarkoittaa ohjelmistoarkkitehtuurissa rajapintaa käyttävää osapuolta. Rajapintaa voi käyttää metodi, ominaisuus tai tapahtuma, ja se välitetään edelleen joko suoraan, muuntamalla tai käärimällä.

Avoin lähdekoodi mahdollistaa sovelluksen käyttäjälle pääsyn alkuperäiseen lähdekoodiin, ja sen muokkaamisen. Avoimen lähdekoodin on oltava erillisen lisenssin alainen, jotta sitä voitaisiin käyttää kaupalliseen tarkoitukseen. Tekijänoikeudet pysyvät alkuperäisen koodin tekijällä.

Dummy-luokka on testauskäyttöön tarkoitettu luokka. Tässä tutkielmassa se on määritelty siten, että se sisältää vain konstruktorin, jossa on pakolliset attribuuttien alustukset, pakolliset get-set-propertyt sekä nolla- tai null-arvoja palauttavat metodit.

Fysiikkamoottori sisältää fysiikan mallintamiseen liittyvään laskentaan tarvittavat ohjelmamoduulit. Fysiikkamoottorin tulisi olla tehokas ja helppokäyttöinen, mutta käytännössä se ei voi olla molempia. Jossakin kirjallisuudessa puhutaan fysiikkakirjastosta fysiikkamoottorin sijaan.

IDE on sovelluskehitysalusta (englanniksi integrated development environment), joka tarjoaa ohjelmoijalle käyttäjäystävällisen kehitysympäristön. Se on yleensä graafinen käyttöliittymä, ja siinä voi olla tuki useammalle kuin yhdelle ohjelmointikielelle (vrt. Visual Studio tai Eclipse).

MIT-lisenssi määrittelee käyttäjälle oikeudet muokata ja levittää sovellusta vapaasti sillä ehdolla, että lisenssin teksti säilyy lähdekooditiedos-

toissa. MIT-lisenssin alaista tuotetta voidaan käyttää kaupallisissa ei-avoimissa lähdekoodeissa.

Pelikehys ja yleensä kehys (englanniksi framework), on kehitetty nopeuttamaan ja helpottamaan tietyssä ympäristössä tapahtuvaa ohjelmointia. Kehys sisältää tavallisimmin jonkin valmiin projektipohjan ja luokkakirjaston.

Pelimoottori sisältää pelin suorittamiseen tarvittavaa ohjelmakoodia, ja se vastaa kuvaruudulle piirtyvän grafiikan päivittymisestä, äänitehosteista sekä kaikesta pelin ja pelaajan välisestä interaktiosta. Jossakin kirjallisuudessa puhutaan pelikirjastosta pelimoottorin sijaan.

Projektiili tarkoittaa lentävää kappaletta, joka on tavallisimmin ampumaaseen ammus. Tietokonepelien yhteydessä projektiilit ovat pienehköjä nopeasti liikkuvia fysiikkaobjekteja, jotka reagoivat fysiikan lakeihin normaalisti, mutta joiden törmäystarkastelu poikkeaa normaaleista fysiikkaobjekteista.

Rajapinta käsittää ohjelmistotekniikassa joukon sellaisia metodeja ja attribuutteja, joita voidaan käyttää jonkin rajatun ohjelmalohkon ulkopuolelta. Rajapintojen avulla pystytään lisäämään ohjelman modulaarisuutta, mikä puolestaan lisää ylläpidettävyyttä.

Sensori tarkoittaa ohjelmistoarkkitehtuurissa sitä osapuolta, jonka ominaisuutta käytetään rajapinnan kautta. Sensorin näkyvyys on tavallisesti julkinen, ja sen käyttö on tarkoin määritelty ja rajattu.

XNA Framework on Microsoftin lanseeraama .Net-ympäristöön kehitetty, peliohjelmointiin tarkoitettu oliopohjainen kehys. Se on ladattavissa ilmaiseksi, ja sen pääasiallinen käyttöympäristö on Microsoft Visual Studio.

Sisältö

Esipuhe **i**

Sanasto **ii**

1 Johdanto **1**

1.1 Työn taustaa 1

1.2 Tutkimusongelma 2

1.3 Tutkimusmenetelmät ja odotetut tulokset 3

2 Kaksiulotteisen pelimaailman fysiikan mallintaminen **5**

2.1 Pelimoottoreiden ja -kehysten toimintaperiaate yleisesti 5

2.2 Fysiikan mallintamisessa käytettyjä kaavoja 6

2.2.1 Kappaleiden mekaniikka 6

2.2.2 Kappaleiden kinematiikka 7

2.2.3 Kappaleiden törmääminen 8

2.3 Törmäystarkastelu 9

2.3.1 Törmäyksen käsittelyn perusteita 9

2.3.2 Törmäävien kappaleiden määrittely 10

2.3.3 Törmäystarkastelun haasteita 12

2.4 XNA-pohjaiset fysiikkamoottorit 14

2.4.1 XNA-pelikehys yleisesti 15

2.4.2 Muutama XNA-pohjainen fysiikkamoottori 16

2.4.3 Itse toteutettu fysiikkamoottori 17

2.4.4 Fysiikkamoottoreiden tarkastelussa tehdyt huomiot 17

3 Adapterikirjaston toteutus **18**

3.1 Adapterikirjaston periaate ja suunnittelumalli 18

3.2 Testaus peleillä 19

3.2.1 Objektin törmäyksen ja törmäyskulman testaaminen 20

3.2.2 Objektiin vaikuttavien voimien manipuloinnin testaaminen . 21

3.2.3 Eri voimien yhteisvaikutuksen hallinnan testaaminen 22

3.2.4 Fysiikkaobjektien välisten akseliliitosten testaaminen 23
3.2.5 Projektiilien testaaminen 25
3.3 Kokeilu- ja kehitysprojektit 26
3.3.1 Kokeiluprojekti 1 27
3.3.2 Kokeiluprojekti 2 27
3.3.3 Kokeiluprojekti 3 28
3.3.4 Kokeiluprojekti 4 28
3.3.5 Kokeiluprojekti 5 29
3.3.6 Kehitysprojekti 1 29
3.3.7 Kehitysprojekti 2 30
3.3.8 Kehitysprojekti 3 31
3.3.9 Kehitysprojekti 4 31
3.3.10 Kehitysprojekti 5 32
3.4 Havaitut ongelmakohdat ja ratkaisut perusteluineen 32

4 Adapterikirjaston rakenne 35
4.1 Adapterikirjaston toimintaperiaate rajapintana 35
4.2 Kokeiluvaiheen staattinen versio 36
4.3 Toteutusvaiheen oliomallinen versio 37
4.4 Jypeli-kirjaston rakenne ennen adapterikirjastoa 38
4.5 Jypeli-kirjaston rakenne adapterikirjaston jälkeen 40
4.6 Adapterikirjaston keskeisimmät luokat 42

5 Johtopäätökset 43
5.1 Tutkielman tavoitteiden toteutuminen 43
5.1.1 Uuden fysiikkamoottorin valinta 43
5.1.2 Uuden fysiikkamoottorin koekäyttöönotto 44
5.1.3 Adapterikirjaston kehittäminen 44
5.2 Adapterikirjaston jatkokehityksen vaihtoehdot 44
5.2.1 Erilliset Jypeli-kirjastot 45
5.2.2 Kehitysprojekteissa toteutetun adapterikirjaston jatkokehitys 45
5.2.3 Jypeli-kirjaston uudelleenkirjoitus 45

6 Projektin aikana esiin tulleet jatkokehitysideat 46
6.1 Boxboy-tyylinen tasohyppelyhahmo 46
6.2 C-kielisiä fysiikkamoottoreita tukeva adapterikirjasto 47

6.3 Rinnakkaisohjelmoinnin tuominen Jypeli-kirjastoon 47
6.4 Raycasting-tekniikkaa käyttävien 3D-pelien kehittely 47
6.5 Useamman pistemäisen painovoimakentän maailma 48

7 Yhteenveto 49

8 Lähteet 50

Liitteet

A Jypeli-kirjasto ennen adapterikirjastoa 52

B Jypeli-kirjasto adapterikirjaston jälkeen 53

C Staattinen Adapteri-luokka 54

D Farseer Physics -adapterikirjaston PhysicsGameBase-luokka 58

E Farseer Physics -adapterikirjaston PhysicsGame-luokka 64

F Farseer Physics -adapterikirjaston PhysicsObject-luokka 66

G Physics2DDotNet-adapterikirjaston PhysicsGameBase-luokka 78

H Physics2DDotNet-adapterikirjaston PhysicsGame-luokka 84

I Physics2DDotNet-adapterikirjaston PhysicsObject-luokka 87

1 Johdanto

Tämän tutkielman tarkoituksena on tarkastella avoimen lähdekoodin fysiikkamoottoreiden käyttöönottoa pelikehityksessä. Tutkielman teoriaosassa tarkastellaan fysiikan ilmiöiden mallintamisen teoriaa. Konstruktiivisessa osassa toteutetaan Jyväskylän yliopistossa kehitetyn Jypeli-kirjaston yhteyteen adapterirajapinta fysiikkamoottoreille. Sen tarkoitus on mahdollistaa eri XNA-kehystä käyttävien avoimen lähdekoodin fysiikkamoottoreiden nopea käyttöönotto ja testaaminen.

1.1 Työn taustaa

Kaksikymmentä vuotta sitten puhuttaessa pelikehityksestä, voitiin yleisesti saada käsitys, että se on raskasta ja vaikeaa. Pelinkehitys tänä päivänä on edelleen raskasta ja vaikeaa. Erona on vain se, että edellä mainitun lisäksi pelien kehittely voi olla nykypäivän ohjelmointityökauluilla samalla hauskaa ja antoisaa. C++ ja etenkin Java toivat oliokeskeisen suunnittelun ja ohjelmoinnin kaikkien saataville. Avoimen lähdekoodin sovelluksia alkoi ilmestymään, ja nopeat laajakaistaiset tietoliikenneyhteydet mahdollistivat niiden levittämisen nopeasti ja vaivattomasti. Peleissä käytetyt fysiikkamoottorit olivat aikaisemmin harvinaisia ja kalliita. Sen lisäksi niiden kehittely edellytti tekijältään syvällistä ohjelmointiosaamista. Tänä päivänä fysiikkamoottoreita on saatavilla avoimena lähdekoodina. C# -kielen ilmestymisen myötä yksi ehkä näkyvimmistä pelikehityksen edistysaskeleesta oli Microsoftin vuonna 2004 julkaisema XNA-työkalusetti, josta myöhemmin kehitettiin XNA Game Studio -sovelluskehitysalusta. XNA Framework, joka on siis XNA Game Studion käyttämä kehys, tarjoaa pelimoottorin, mutta ohjelmoijan on itse ohjelmoitava tai hankittava siihen sopiva fysiikkamoottori. Tämä on perusteltua sillä, että fysiikkamoottorilta vaaditut ominaisuudet muovautuvat vallitsevan tarpeen mukaan. On pelejä, joissa vaaditaan tarkkaa, mutta hidastempoista fysiikan laskentaa, sekä pelejä, joissa vaaditaan taas nopeaa laskentaa tarkkuudesta tinkien. Joissain fysiikkamoottoreissa on mahdollista myös säätää näitä ominaisuuksia.

Nuorten peliohjelmointikurssilla käytetty pelikirjasto Jypeli kehitettiin kehykseksi XNA-pelikehyksen päälle. Jypelin idea on tehdä pelikehityksestä niin helppoa, että kolmetoistavuotias nuori voisi sen oppia. Siinä tapahtumien käsittely, sekä piirrot ja päivitykset hoidetaan siten, että ohjelmoijan ei tarvitse niistä tietää. Käytännössä ohjelmointi Jypeli-kirjastoa käyttäen on sitä, että määritellään mitä mikäkin on, ottamatta kantaa algoritmeihin. Ideologialtaan se muistuttaa hieman funktionaalista ohjelmointitapaa. Jypeli-kirjasto käyttää fysiikkamoottorina avoimen lähdekoodin ja MIT-lisenssin alaista Physics2DDotNet fysiikkamoottoria. Muita mahdollisia XNA:n päällä toimivia valmiita fysiikkamoottoreita ovat muun muassa Oops!, Farseer, Box2D.XNA, JigLibX, Jitter Physics ja BEPU Physics. Osa niistä on vielä betatestausvaiheessa.

1.2 Tutkimusongelma

Jypeli-kirjasto tarjoaa nuorille pelinkehittäjille esimääritellyn valikoiman erilaisia peligenrejä. Niitä on saatavilla valmiina projektipohjina aina tasohyppelyistä autopeleihin. Varsin suuressa osassa pelejä joudutaan pelimaailmassa kamppailemaan fysiikan voimia vastaan. Nuoresta iästään johtuen Jypeli-kirjasto on vielä keskeneräinen. Jypeli-kirjaston ylläpidon aikana on havaittu, että osa peleissä ilmenneistä ohjelmavirheistä viittaa fysiikkamoottorin lähdekoodiin. Vaikka Jypelin käyttämä Physics2DDotNet fysiikkamoottori on MIT-lisenssin alainen, on kehitystiimissä katsottu parhaaksi olla tekemättä muutoksia itse fysiikkamoottorin lähdekoodiin. Ongelmatilanteet fysiikkamoottorin toiminnallisuudessa ovat ratkaistu erilaisilla kiertokonsteilla ja tilapäisratkaisuilla.

Toisen ylläpitämästä lähdekoodista on työlästä ja vaikeaa paikantaa virheitä. Nopeampi keino voisi olla kokeilla jotain toista fysiikkamoottoria. Monet fysiikkamoottoreiden viat saattavat olla todellisuudessa ominaisuuksia, jotka saattavat oikealla hetkellä toimia jopa pelikehittäjän eduksi. Näin ollen voisi olla hyödyllistä pystyä vaihtamaan fysiikkamoottoria nopeasti, tai jopa käyttämään useampaa yhtäaikaa siten, että jokaiselle peligenrelle olisi oma, siinä parhaiten toimivaksi katsottu fysiikkamoottori. Jypeli-kirjaston ja siinä käytetyn fysiikkomoottorin välisen kytkösasteen toteamiseksi poistettiin koeluontoisesti Jypeli-kirjastosta kokonaan fysiikkamoottori, ja kirjattiin kääntäjän antama palaute. Siitä ilmeni, että kymmenessä Jypeli-kirjaston luokassa oli yhteensä 37 viitettä 28:aan eri Physics2DDotNet fysiikkamoottorin luokkaan. Voidaan siis todeta Jypeli-kirjaston ja fysiikkamoottorin välisen kytkösasteen olevan korkeahko.

Fysiikkamoottorin suoraan vaihtaminen veisi viikkojen tai jopa kuukausien työtunnit yhdeltä kokeneelta ohjelmoijalta. Jotta fysiikkamoottorin vaihtaminen jatkossa olisi mielekästä, olisi tarpeen erottaa Jypeli-kirjasto ja fysiikkamoottori selkeästi toisistaan. Tässä tutkielmassa tutkitaan toteutusratkaisuna adapterikirjaston toteuttamista edellä mainittujen kirjastojen väliin. Haasteena on olemassa olevien XNA-pohjaisten avoimen lähdekoodin fysiikkamoottoreiden erilaisuus. Joka tapauksessa on luovuttava siitä ajatuksesta, että voitaisiin luoda yksi kirjasto, joka toimisi kaikkien fysiikkamoottoreiden kanssa. Kyseeseen tulisi joko jokaiselle fysiikkamoottorille toteutettu oma adapterikirjasto, tai vaihtoehtoisesti yksi kirjasto, johon voitaisiin pienehköllä vaivalle luoda lisäosa uutta fysiikkamoottoria varten. Tärkeintä adapterikirjastossa olisi erottaa selkeästi kolme sisäistä osaa. Ne olisivat Jypelin kanssa keskusteleva osa, fysiikkamoottorin kanssa keskusteleva osa ja tulkkiosa joka sijoittuisi näiden kahden edellisen osan välille. Näin toimien voitaisiin vaikuttaa myönteisesti tulevan adapterikirjaston lähdekoodin ylläpidettävyyteen.

1.3 Tutkimusmenetelmät ja odotetut tulokset

Tutkimustyössä voidaan erottaa toisistaan kaksi selkeää asiakokonaisuutta, kvalitatiivinen ja empiiris-konstruktiivinen osio. Kvalitatiivisessa osiossa käsitellään fysiikkamoottoreihin liittyvää teoriaa, ja yritetään sitä kautta saavuttaa ymmärrys tutkittavaan asiaan. Aineistoa kerätään erilaisista fysiikkamoottoreita käsittelevistä kirjoista sekä internet-julkaisuista. Konstruktiivis-empiirisessä osiossa kehitetään adap-

terikirjasto ja tarkastellaan siihen kytkettyjen fysiikkamoottoreiden käyttöönottoa. Adapterikirjasto kehitetään liittäen siihen samalla uusi fysiikkamoottori. Saatujen tulosten perusteella voidaan todeta muiden fysiikkamoottoreiden liittämisen mielekkyys. Samalla on myös pystyttävä osoittamaan liitetyn fysiikkamoottorin toimivuus esimerkkipelien avulla. Käytettävien esimerkkipelien määrää kasvatetaan kehittelyn edetessä ja käyttöön otettavien ominaisuuksien kasvaessa.

Ensisijainen tavoite tässä tutkielmassa on saada varmuus siitä, että adapterikirjaston kehittäminen ja käyttöönotto on mielekästä. Mikäli todetaan, että adapterin kautta käyttöön otettu toinen fysiikkamoottori ei tuo parannuksia fysiikan mallinnukseen peleissä, voidaan adapterikirjasto jättää toteuttamatta. Silloin tutkielmassa keskitytään tarkastelemaan käyttöönoton hankaluuksien syitä. Tällaisia syitä voivat olla esimerkiksi fysiikkamoottorien liian suuret keskenäiset eroavaisuudet luokkarakenteissa. Onnistuneen käyttöönoton seurauksena tehdään vertailevaa tutkimusta vanhan ja uuden fysiikkamoottorin välillä vahvistaen siten adapterin jatkokehittelyn perustelua. Samalla kartoitetaan sitä työmäärää, mitä fysiikkamoottorin vaihtaminen adapterikirjastoon edellyttää, ja mitä asioita täytyy uudesta käyttöön otettavasta fysiikkamoottorista selvittää.

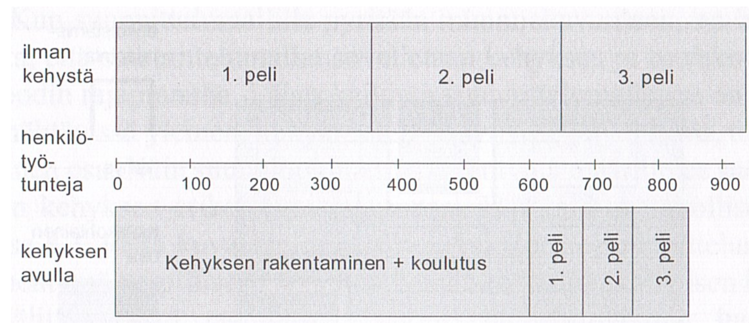
Odotettuna ja mielekkäimpänä tuloksena tutkielmassa olisi siis todeta fysiikkomoottorien kohtuullisen vähävaivainen käyttöönotto sekä todeta Jypeli-kirjaston fysiikkaominaisuuksien selvä paraneminen. Fysiikkamoottorin vaihdon helpottamisella saavutettaisiin Jypeli-kirjaston parempi ylläpidettävyys siinäkin tapauksessa, että jonkin siinä käytetyn fysiikkamoottorin ylläpito lakkaisi kokonaan, tai että lisenssiehdot muuttuisivat yllättäen. Myös oman fysiikkamoottorin kehitys ja käyttöönotto voisi helpottua oleellisesti.

2 Kaksiulotteisen pelimaailman fysiikan mallintaminen

Pelien ohjelmointi on työlästä. Etenkin vanhat pelit, joissa käsitellään grafiikkaa ja ääntä, ovat osittain koodattu Assemblerillä [10, sivu 7]. Vaikka tekniikka on kehittynyt huimasti vuosien saatossa, ovat tietyt perusperiaatteet pysyneet täysin samana. Tekniikan kehittyminen näkyy pelikehityksessä siinä, että peleiltä vaaditaan kykyä mallintaa tosimaailmaa entistä tarkemmin. Tässä luvussa käsitellään yleisellä tasolla fysiikan mallintamista peliympäristössä. Huomionarvoista on se, että tietokone joutuu laskemaan fysiikan kaavoja useita kertoja sekunnissa. Näin ollen kaavat eivät saa olla liian raskaita ja monimutkaisia ratkaistaviksi, ettei pelikokemus kärsisi tietokoneen suoritusnopeuden hidastumisesta.

2.1 Pelimoottoreiden ja -kehysten toimintaperiaate yleisesti

Pelimoottorit ovat erikoistapaus kehyksistä. Kehyksen periaate on tarjota kehittäjälle sellainen kehitysympäristö, jossa on ikään kuin tyhjiä aukkoja, jotka täyttämällä kehittäjä pystyy rakentamaan sovelluksensa [8, sivu 187]. Kehyksen käyttäminen tulee kyseeseen silloin, kun ollaan tekemässä useampaa sellaista sovellusta, joissa on riittävästi samoja ominaisuuksia kehyksen rakentamista varten. Kuvan 2.1 mukaan jo kolmannen peli kehittämisen jälkeen kehyksen rakentaminen on osoittautunut kannattavaksi.



Kuva 2.1: Työtuntimäärien vertailu pelisovellusten tapauksessa [8, sivu 189].

Erityispiirre pelikehyksissä ja peliohjelmoinnissa yleisesti verrattuna perinteiseen ohjelmointiin on se, että peliä suoritetaan jatkuvasti. Toisin sanoen, siinä missä perinteisen tietokoneohjelmat siirtyvät tilasta toiseen ja lopuksi antavat tuloksen, pelimoottorit ovat niin kutsutussa ikuisessa silmukassa. Jokaisella kierroksella tarkastetaan peliobjektien sijainnit, päivitetään ne ja piirretään uudestaan kuvaruudulle. Fysiikkamoottori toimii sillä perusteella, että sille annetaan ensiksi peliobjekti, joka sisältää attribuutteina esimerkiksi massan, kitkan ja muodon. Sen jälkeen fysiikkamoottorille kerrotaan, että objektiin vaikuttaa jonkin vektorin suuntainen voima. Tämän jälkeen fysiikkamoottori laskee ja määrittää fysiikkaobjektin uuden sijainnin koordinaatistossa sille annetun aikaintervallin perusteella.

2.2 Fysiikan mallintamisessa käytettyjä kaavoja

Tietokoneet mallintavat tosimaailmaa laskemalla erilaisia fysiikan kaavoja. Mallinnus ei koskaan tule olemaan täydellistä, mutta nykyajan laitteilla päästään jo hämmästyttävän lähelle todellisuutta. Vaativimpia fysiikan mallinnuksen kohteita ovat kolmiulotteista grafiikkaa käyttävät pelit, joissa on fysiikan lisäksi pikkutarkkaa pintatekstuuria sekä valojen heijastumista ja varjojen muodostumista. Fysiikkaa mallinnetaan tietokoneessa vektoreilla, ja vektoreita lasketaan matriisien avulla. Seuraavaksi tarkastellaan fysiikan kaavoja, joita esiintyy esimerkiksi yksinkertaisen tasohyppelypelin toteutuksen yhteydessä.

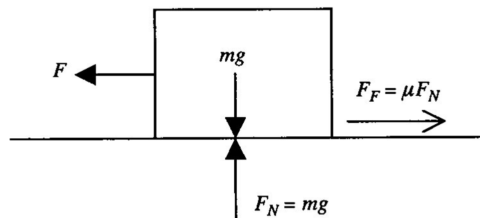
2.2.1 Kappaleiden mekaniikka

Mekaniikan laeista ehkä yleisin mallinnettava on painovoima ja putoamiskiihtyvyys. Kun kerran tiedetään, että putoamiskiihtyvyys on kaikille massoille vakio, tarvitaan vain ottaa huomioon ilmanvastus tai muu vastaava kitkatekijä. Peleissä, joissa liikutellaan erisuuruisia massoja, on otettava huomioon inertia. Avaruusaiheisissa peleissä taas saatetaan joutua simuloimaan kahden kappaleen vetovoimaa gravitaatiolain mukaisesti.

$$F_g = \frac{Gm_1m_2}{r^2} \tag{2.1}$$

Kaavassa (2.1) G on gravitaatiovakio, jonka likiarvo on $6,67428 * 10^{-11} Nm^2kg^{-2}$. Toisiinsa vaikuttavien kappaleiden massat ovat m_1 ja m_2. Kappaleiden välistä etäi-

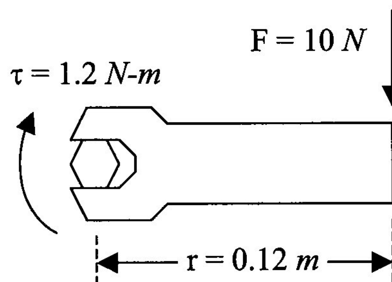
syyttä kuvaa r [14, sivu 25]. Toinen hyvin merkittävä ja peleissä huomioon otettu mekaniikan seikka on kitkavoima. Kuvassa 2.2 F_F on kitkavoima, joka vaikuttaa voiman F vastakkaissuuntaisesti. Kitkavoiman ja inertian tarkastelua yhdistelemällä voidaan esimerkiksi autopeleissä viedä fysiikan mallintaminen todella syvälle. Sen sijaan, että tarkastellaan auton liikkumista kilparadalla pelkkänä suorakaiteen muotoisena runkona, voidaan tarkastella vetävissä ja ei-vetävissä renkaissa tapahtuvia fysiikan ilmiöitä.



Kuva 2.2: Kitkavoimat kappaleessa [14, sivu 33].

2.2.2 Kappaleiden kinematiikka

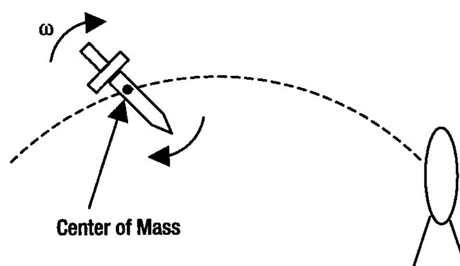
Kinematiikka käsittää nopeuden, kiihtyvyyden ja paikan tarkastelua. Kappaleisiin vaikuttavia voimia jaetaan x- ja y -akseleiden suuntaisiksi komponenteiksi, jos tarkastelu tapahtuu kaksiulotteisessa avaruudessa. Kolmiulotteisessa avaruudessa voimat jaetaan vastaavasti kolmeen komponenttiin. Samaa tarkastelua otetaan mukaan myös pyörivän liikkeen tarkasteluun. Pelkän pyörimisnopeuksien ja kiihtyvyyksien lisäksi kinematiikassa tarkastellaan myös vääntömomenttia 2.3.



Kuva 2.3: Voima vipuvarressa saa aikaan momentin [14, sivu 71].

Peliohjelmoinnissa on viisasta lähteä siitä oletuksesta, että kaikki kappaleet ovat jäykkiä. Edistyneemmällä tasolla voidaan kappaleet koostaa partikkeleista simuloi-

den siten ei-jäykkyyttä. Samassa yhteydessä voidaan myös hoitaa kappaleen hajoaminen partikkeleihin esimerkiksi törmäystilanteessa. Momentin käyttäminen tulee esiin pelitilanteessa tavallisimmin autopeleissä, joissa momentin avulla määritellään pyörää pyörittävä voima. Kaikilla jäykillä kappaleilla on staattinen massakeskipiste. Jos kappale pyörii massakeskipisteensä ympäri, se pyörii silloin keskeisesti, eikä poikkea massakeskipisteen suhteen lentoradaltaan 2.4.



Kuva 2.4: Massakeskipisteen lentorata tikarin viuhuessa kohti maalia [14, sivu 74].

Massakeskipisteen tarkastelu voi tulla kyseeseen siinä tapauksessa, mikäli pelissä aiheutetaan törmäyksiä monimutkaisesti muotoilluille dynaamisille peliobjekteille, ja odotetaan niiden käyttäytyvän oikeiden fysiikan lakien mukaisesti.

2.2.3 Kappaleiden törmääminen

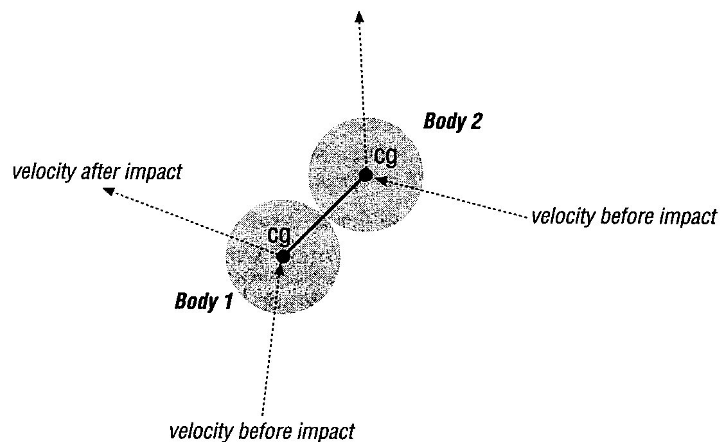
Törmäyksen teoriassa fysiikan kannalta oleellisin asia on liikemäärän säilymislaki (2.2). Törmäykset jaotellaan pääsääntöisesti kahteen ryhmään, kimmottomiin ja kimmoisiin. Täysin kimmoisessa törmäyksessa staattiseen kappaleeseen törmätessä kappaleen nopeus säilyy, ja täysin kimmottomassa törmäyksessä liike pysähtyy törmäyksen seurauksena. Kimmoisuus määritellään kimmoisuuskertoimella ϵ, joka on suljetulla välillä 0–1 (2.3). Todellisuudessa täysin kimmottomassakin törmäyksessä liikemäärä säilyy, mutta objektin liikuttamisen sijaan liikemäärä kuluu kappaleen sisäiseen muodon muuntumiseen. Tietokonemaailmassa liikemäärää hävitetään kimmoisuuskertoimen edellyttämä määrä, ja jäykkiä kappaleita mallinnettaessa muodonmuutoksia ei huomioida. [17, sivut 379–383]

$$m_1 v_{1i} + m_2 v_{2i} = m_1 v_{1f} + m_2 v_{2f} \tag{2.2}$$

$$(v_{1f} - v_{2f}) = -\epsilon(v_{1i} - v_{2i}) \tag{2.3}$$

2.3 Törmäystarkastelu

Pelimaailmassa fysiikan mallintaminen olisi melko puutteellista, ellei törmäystarkastelua otettaisi huomioon. Itse asiassa kappaleet putoaisivat toistensa läpi eikä näin ollen mikään dynaaminen kappale pystyisi olemaan staattisesti pelimaailmaan kiinnitettyjen objektien päällä. Fysiikan näkökulmassa törmäystarkastelu tutkii lähinnä toisiinsa törmäävien kappaleiden tulo- ja lähtökulmia 2.5. Tietokonemaailmassa törmäys on siis tapahtuma, joka aktivoi aliohjelmat laskemaan fysiikan kaavojen avulla törmäyksen seurauksia. [2, sivu 87] Törmäyksen havaitsemisessa jou-



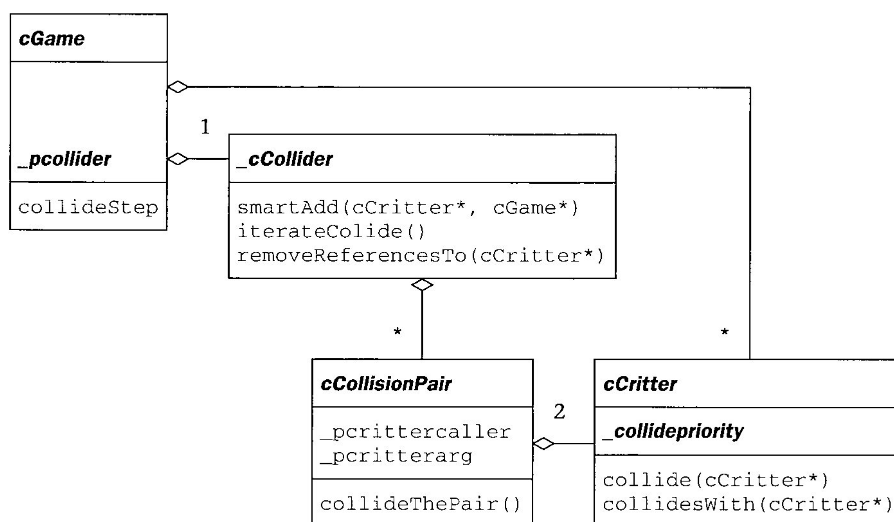
Kuva 2.5: Kaksi törmäävää ympyränmuotoista objektia [2, sivu 95].

dutaan turvautumaan siihen tarkoitukseen erikoistettuihin algoritmeihin. Mikäli pelissä olisi vain muutama törmäävä objekti, ei olisi raskasta havaita niiden törmäyksiä. Tilanne muuttuu toiseksi, kun törmäileviä objekteja on ruudulla satoja, ellei tuhansia. Eräs ratkaisu tähän on jakaa pelialue isoihin ruutuihin, ja tarkastella jokaista ruutua kerrallaan sen sisällä olevien objektien suhteen. Mikäli yhdessä ruudussa on kaksi objektia, voidaan niiden olettaa olevan törmäysalttiita, ja aloittaa kyseisten objektien törmäystarkastelu. Kun objektien määrä kasvaa, voidaan kasvattaa ruutujen määrää sekä syventää tutkittavaa hakupuuta.

2.3.1 Törmäyksen käsittelyn perusteita

Kahden fysiikan kappaleen törmäyksessä kysymys on siis siitä, että niiden geometriat menevät sisäkkäin. Ympyränmuotoisilla kappaleilla se tarkoittaa yksinkertaisimmillaan sitä, että keskipisteiden välinen etäisyys on pienempi, kuin molempien kappaleiden säteiden summa. Kantikkailla kappaleilla laskenta on hieman moni-

mutkaisempi, mutta periaate on sama. Useamman objektin maailmassa törmäykset on käytävä läpi siten, että tutkitaan kappale kerrallaan, törmääkö jokin kappale toiseen vai ei. Algoritmisesti on ratkaistava se, että kappaleen törmäämistä itseensä ei huomioida, eikä jo kerran tarkasteltua törmäysparia tarkastella uudestaan toisen parin kannalta.



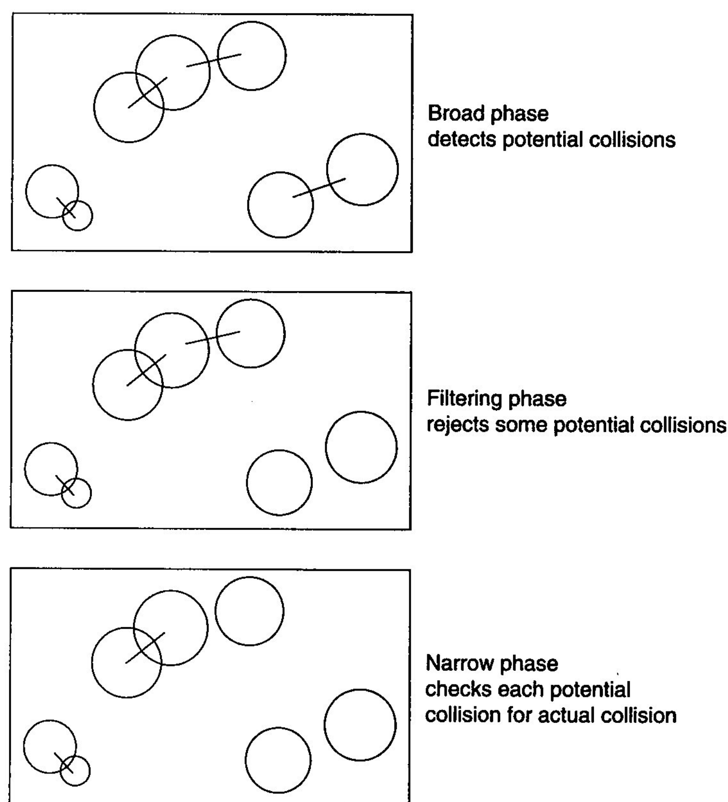
Kuva 2.6: Törmäysarkkitehtuuri [16, sivu 245].

Törmäyksen arkkitehtuureja voidaan määritellä monella eri tavalla sovelluskohtaisesti. Kuvassa 2.6 on esitetty eräs näkemys törmäyksenkäsittelyn arkkitehtuurista. Kuvassa esiintyvä `Critter` on synonyymi fysiikkaoliolle. `Collidepriority`-attribuutin avulla voidaan priorisoida eri törmäyksiä sen suhteen, että kumman törmäysparin kannalta törmäystä tarkastellaan. Priorisoitavia fysiikkaobjekteja voivat olla mm. seinät, projektiilit, pelaaja sekä muut fysiikkaobjektit. [16, sivut 244–246]

2.3.2 Törmäävien kappaleiden määrittely

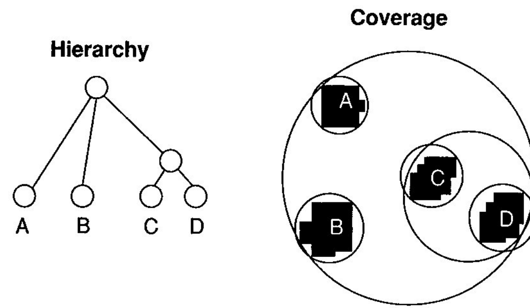
Varsinaisesta törmäyksestä puhutaan siis vasta silloin, kun kaksi kappaletta ovat törmäysalueeltaan joko osittain sisäkkäin, tai törmäysalueen rajaviivat leikkaavat toisiaan. Törmäysalue on yleensä jokin yksinkertainen perusmuoto, esimerkiksi ympyrä tai nelikulmio, jonka sisällä varsinainen fysiikkaobjekti on. Fysiikkaobjekti on siis tavallisimmin jokin hahmo tai ajoneuvo. Tarvittaessa voidaan törmäysalue määrittää käsittämään varsinaisen fysiikkaobjektin ääriviivat. Se tekee törmäystarkastelusta haastavampaa, eikä arcade-tyylisissä peleissä ole tarvetta niin tarkkaan tör-

mäysten käsittelyyn. Törmäystapahtumaa ennen kappaleita tutkitaan ja käsitellään mahdollisina törmääjinä. Teoriassa kaikki 2D-maailmassa olemassa olevat dynaamiset kappaleet ovat keskenään mahdollisia törmääjiä. Kuitenkin käytännössä ollaan kiinnostuneita siitä, mitkä kappaleet ovat todellakin törmäämässä toisiinsa. Tätä rajausta voidaan tehdä esimerkiksi määrittelemällä jokin kappaleiden välinen minimietäisyys, jolloin ollaan kiinnostuneita siitä, törmäävätkö kappaleet toisiinsa.



Kuva 2.7: Kolmivaiheinen törmäystarkastelu [12, sivu 256].

Kuvassa 2.7 esitellään kolmivaiheinen törmäystarkastelu, jossa ensimmäisessä vaiheessa etsitään toisiaan lähellä olevat kappaleet. Toisessa vaiheessa hylätään jotkin törmäyskandidaateista. Hylkäämisen syy voi olla esimerkiksi se, että kyseiset kappaleet loittonevat toisistaan. Viimeisessä vaiheessa tutkitaan kappaleet, joiden törmäysalueen ääriviivat leikkaavat. Fysiikkaobjektien sijaintitieto pidetään yllä puumaisessa rakenteessa, jossa lehtisolmut ovat yksittäisiä fysiikkaobjekteja, ja isäntäsolmut lähekkäin sijaitsevia fysiikkaobjekteja. Fysiikkaobjekteja voidaan rajata hierarkisesti kolmella eri tavalla, alhaalta ylös, ylhäältä alas sekä insertiolla 2.8.



Kuva 2.8: Hierarkinen törmäystarkastelu [12, sivu 260].

Alhaalta ylöspäin suuntautuneessa tarkastelussa haetaan ensin yksittäiset fysiikkaobjektit, joista tulevat tarkastelupuun lehtisolmut. Lähekkäin olevat lehtisolmut yhdistetään rajaamalla ne yhteisellä törmäystarkastelualueella, ja tarkastelupuussa siitä alueesta tulee sisällä olevien objektien solmujen isäntäsolmu. Näin jatketaan, kunnes kaikki solmut ovat yhden ainoan törmäystarkastelualueen sisällä, ja siitä alueesta tulee tarkastelupuussa juurisolmu. Ideaalisessa tilanteessa tarkastelupuu on tasapainoinen. Ylhäältä alaspäin suuntautuneessa tarkastelussa toimitaan juuri päinvastoin rajaamalla ensin kaikki fysiikkaobjektit yhden tarkastelualueen sisälle, ja rajaamalla edelleen lähekkäin olevat objektit oman tarkastelualueen sisälle kunnes kaikki yksittäiset objektit ovat lehtisolmuja. Insertiossa lisätään tarkasteluun objekteja yksi kerrallaan, ja jokaisen lisäyksen jälkeen kasvatetaan kaikkia objekteja ympyröivää tarkastelualuetta. Mikäli lisättävät objektit ovat lähellä toisiaan, ne ympäröidään yhteisellä tarkastusalueella ja siitä alueesta tehdään isäntäsolmu tarkastelupuuhun. [12, sivut 267–269]

2.3.3 Törmäystarkastelun haasteita

Niinkuin jo edellä todettiin, fysiikan ilmiöiden mallinnuksessa tietokonemaailmassa joudutaan valitettavan usein turvautumaan erilaisiin kompromisseihin. Nykyaikainen tietokonearkkitehtuuri asettaa omat rajansa, ja ohjelmistotekniikan tekninen osaaminen omansa. Tässä tutkielmassa joudutaan tulemaan toimeen yhtäaikaisuuden, tasaisesti levittäytyneiden runsaslukuisten objektien, koveruuden, suurten nopeuksien, objektien ohjaamisen nopeusvektorien avulla sekä ennalta arvaamattomien kimpoamiskulmien aiheuttamien ongelmien kanssa.

Yhtäaikaisuuden ongelma havainnollistuu sellaisessa 2D-fysiikkapelissä, jossa on ladottuna päällekkäin suorakaiteen muotoisia laatikoita siten, että niiden massa-

keskipisteet ovat samalla suoralla. Teoriassa tietokoneen fysiikkamaailmassa eivät vaikuta ilman kierto eikä coriolis-voimat, ja neliömuoto on sataprosenttisen täydellinen. Kuitenkin tätä ilmiötä mallinnettaessa päällekkäin pinotut laatikot sortuvat ennen pitkää ilman minkään ulkopuolisen voiman vaikutusta. Ilmiö syntyy siitä, että kaikkia pintojen välisiä törmäyksiä ei ole mahdollista käsitellä yhtäaikaisesti. Näin ollen yksittäisten laatikoiden minimaalinen asemankorjaus aiheuttaa tornissa huojuntaa, joka johtaa lopulta sortumiseen.

Törmäystarkastelupuuta muodostettaessa lähekkäin sijaitsevat objektit muodostivat aina yhteisen isäntäsolmun. Mikäli objekteja on runsaasti, eli useita satoja, ja ne ovat levittäytyneina tasaisesti, syntyy tilanne, jossa tarkastelupuu ei saa riittävästi syvyyttä, vaan kaikkia törmäyspareja joudutaan tarkastelemaan tasa-arvoisesti. Käytännössä tämä ilmenee pelin hidastumisena ja katkonaisina objektien siirtymisinä. Fysiikkamoottoreissa on useimmiten erilaisia optimointimenetelmiä tällaisten varalle, ja usein esimerkiksi projektiilit käsitellään jollakin poikkeavalla tavalla muihin fysiikkaobjekteihin verrattuna. Tilannetta voidaan helpottaa pelkistämällä tarkasteltavien kappaleiden törmäysalueet esimerkiksi ympyrän muotoon.

Toisinaan halutaan muodostaa törmäyskappaleita suoraan jonkin itse piirretyn epäsäännöllisen geometrian mukaan. Mikäli kappaleen törmäysaluetta pelkistetään johonkin yksinkertaiseen muotoon, saatetaan saada siten aikaiseksi painokeskipisteen ajautumista väärään kohtaan aiheuttaen näin ei toivottua fysiikan käyttäytymistä. Tällaiseen tilanteeseen ajaudutaan yleensä koverien kappaleiden kanssa. Fysiikkamoottoreissa on usein sisäänrakennettuja algoritmejä tällaisten fysiikkaobjektien varalle. Käytännössä koverat kappaleet hajotetaan useaksi fysiikkaobjektiksi, jotka ovat liitettyinä toisiinsa. Liitoksen käytännön toteutuksessa on monta erilaista tapaa, ja tärkeintä on jättää toisiinsa liitettyjen kappaleiden keskenäiset törmäykset huomiotta.

Autopeleissä, tasohyppelypeleissä ja muissa maapallon pinnalle sijoitetuissa pelimaailmoissa nopeudet pysyvät yleensä kohtuullisina. Ammukset, jotka lentävät nopeammin, kuin mitä pelaaja pystyisi havaitsemaan, käsitellään tavallisimmin säteinä, jotka törmäävät johonkin objektiin. Avaruuspeleissä ei sen sijaan ole mitenkään harvinaista tilanne, jossa kappaleet saavat huimia nopeuksia. Ongelmia alkaa ilmenemään, mikäli nopeudet kasvavat niin suuriksi, että törmäävät kappaleet menevät

toistensa läpi. Tällaisessa tilanteessa on kyse siitä, että pelimoottorin päivityskierrokseen kuluneessa ajassa kappale on matkustanut pidemmän matkan, kun mitä törmäyksen kohteena oleva kappale on leveydeltään. Tätä ongelmaa kierretään tavallisesti huippunopeuksia rajoittamalla.

Fysiikkamoottorit ovat optimoituja nimenomaan kappaleeseen kohdistuvien fysiikan ilmiöiden mallintamiseen. Tämä tarkoittaa käytännössä sitä, että kappaleen nopeus ei muutu, mikäli siihen ei vaikuta mikään voima. Vakionopeuden aikaansaamiseksi kappaleelle annetaan impulssi, eli lyhytaikainen voimasysäys liikkeen aikaansaamiseksi, ja sen jälkeen siihen kohdistetaan kitkavoiman suuruinen vastakkaissuuntainen voima. Kuitenkin monet fysiikkamoottorin antavat muuttaa suoraan nopeusvektoria, mikä saattaa saada aikaan odottamattomia ilmiöitä. Tällainen ilmiö voi syntyä silloin, kun fysiikkaobjektia ajetaan jollakin nopeudella staattista fysiikkaobjektia vasten. Seurauksena voi olla objektien meneminen sisäkkäin, tai jokin muu vastaava odottamaton ilmiö.

Ympyrää lukuunottamatta fysiikkaobjektien muodot ovat fysiikkamoottorin kannalta monikulmioita, eli polygoneja. Joissakin fysiikkamoottoreissa jopa ympyrätkin saatetaan mallintaa polygoneiksi, jotka sisältävät niin monta särmää, että sen käytös 2D-fysiikkamaailmassa muistuttaa täydellisen ympyrän käytöstä. Kuitenkin joissakin törmäystapauksissa saattavat terävät kulmat osua toisiinsa, ja sen seurauksena objektien kimpoamiskulmat saattavat poiketa odotetusta. Samaa epätarkkuutta esiintyy toki muissakin muodossa, mutta käytännössä ympyrän muotoisten objektien fysiikan lakien mukaisesta käyttäytymisestä ollaan kaikkein kriittisimpiä. Ympyrä on kuitenkin käsitteenä niin yksinkertainen, ettei sen mallintaminen sellaisenaan pitäisi olla kynnyskysymys fysiikkamoottorin kehittäjälle.

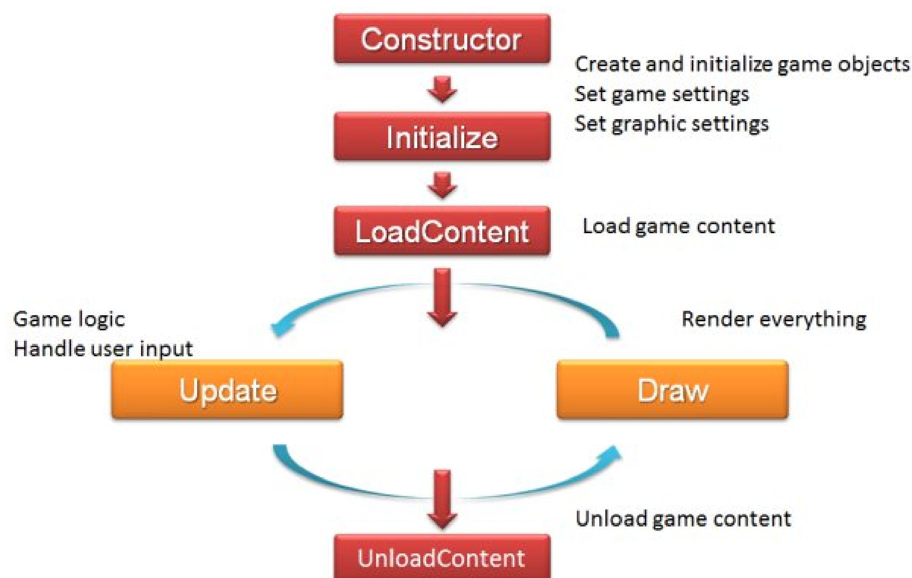
2.4 XNA-pohjaiset fysiikkamoottorit

Aikaisemmin pelikehittäjät joutuivat kehittämään itse omat pelikehyksensä, ja useassa tapauksessa tänäkin päivänä se on järkevää. Kuitenkin vuonna 2004 Microsoft julkaisi XNA-pelikehyksen, joka toi peliohjelmoinnin aivan uudella tavalla harrastajien ulottuville. XNA-pelikehyksen voi ladata itselleen täysin ilmaiseksi, ja sillä tehtyjä pelejä voi julkaista erityisellä Microsoftin lanseeraamalla XNA Creators Club -sivustolla. Tämän tutkielman rajaamiseksi onkin järkevää tyytyä tarkastelemaan

tässä luvussa vain XNA-pelimoottoria tukevia avoimen lähdekoodin fysiikkamoottoreita. Viimeisenä tarkastellaan mahdollisuutta oman itse tehdyn fysiikkamoottorin toeuttamiseksi.

2.4.1 XNA-pelikehys yleisesti

XNA-pelikehyksen toimintaperiaate perustuu puolivalmiiseen pelimoottoriin, jossa on valmiina metodit `Initialize`, `LoadContent`, `UnloadContent`, `Update` ja `Draw` 2.9. `Initialize` hoitaa pelin alustuksen pakolliset rutiinit, eikä peliohjelmoijan tarvitse yleensä puuttua tähän metodiin. Muissa metodeissa on myös valmiina joitain pakollisia toimenpiteitä, mutta koodin ajaminen tuottaa tulokseksi vain tyhjän peli-ikkunan. Näihin metodeihin oikeat toimenpiteet lisäämällä saadaan aikaiseksi toimiva peli. `LoadContent` lataa pelin mediakomponentit, kuten esimerkiksi kuva- ja äänitiedostot, muistiin pelin suoritusta varten. XNA-pelikehys tukee kaikkia yleisimpiä mediaformaatteja. `UnloadContent` nimensä mukaisesti vapauttaa ladatut mediakomponentit muistista pelin päätyttyä. `Update` ja `Draw` -metodia kutsutaan joka kerta pelisilmukkaa läpi käytäessä. Kyseiset metodit saavat parametrinä sen ajan millisekunteina, mikä on kulunut edellisestä kierroksesta. XNA-pelikehyksen pelisilmukka käydään oletusarvoisesti läpi 60 kertaa sekunnissa [11, sivut 29–40].



Kuva 2.9: XNA-pelikehyksen rakenne [19].

Jyväskylän yliopiston toteuttama Jypeli-kirjasto on siis XNA-pelikehyksen päälle toteutettu kehys, jonka tarkoituksena on entisestään helpottaa peliohjelmointia [9, sivu 55].

2.4.2 Muutama XNA-pohjainen fysiikkamoottori

Physics2DDotNet on MIT-lisenssin alainen, Jypeli-kirjastoon alunperin valittu fysiikkamoottori. Se on tarkoitettu käytettäväksi 2D-peleissä, mutta sisältää tuen myös 3D-peleihin. Jypeli-kirjasto ei toistaiseksi tue 3D-pelien tekemistä. Physics2DDotNet on saatavilla versiona 2.0.0.0, mutta päivityksiä lähdekoodiin ei ole tehty vuoden 2010 jälkeen [15].

Oops!-fysiikkamoottori on Microsoft Permissive Licence:n alainen, 3D-peleihin optimoitu fysiikkamoottori. Sen kehitys on vielä betatestausvaiheessa, ja nykyinen saatavilla oleva versio on 0.7. Sen käytölle 2D-peleissä ei pitäisi olla mitään estettä, jos jätetään yksi suuntavektoreista huomioimatta. Keskustelupalstalla viimeisin kommentti on vuoden 2012 helmikuulta, ja viimeisin lähdekoodin julkaisu on tehty vuonna 2009 [13].

JigLibX on MIT-lisenssin alainen avoimen lähdekoodin fysiikkamoottori 3D-pelien kehittämiseen. Kotisivuilla nähtävät videoleikkeet ovat vaikuttavia, ja sen käyttöön 2D-peleissä ei pitäisi olla esteitä. JigLibX on betatestausvaiheessa versionumerolla 0.3.1, joka on julkaistu vuonna 2009 [4].

Box2D on zlib-lisenssin alainen, nimensä mukaisesti 2D-peleihin erikoistunut fysiikkamoottori. Tällä hetkellä julkaistuna on versio 2.2.1 vuodelta 2011, ja keskustelufoorumista löytyy merkintöjä vuodelta 2012. Tästä voidaan päätellä, että Box2D-fysiikkamoottorin kehitys jatkuu vilkkaana ainakin toistaiseksi [3].

Farseer Physics on Box2D:n pohjalta kehitetty fysiikkamoottori Microsoft Permissive Licence:n alainen, 2D-peleihin tarkoitettu fysiikkamoottori. Farseer Physics on päivitetty versioon 3.3.1 vuonna 2011. Keskustelupalstalta löytyy erittäin tuoreita merkintöjä vuoden 2012 puolelta. Tämä seikka sekä pikainen katsaus harrastajien keskustelupalstoille puoltaa osittain tämän fysiikkamoottorin valintaa ensimmäiseksi Jypeli-kirjaston vaihdosfysiikkamoottoriksi [20].

2.4.3 Itse toteutettu fysiikkamoottori

Jypeli-kehitystyöryhmän kanssa tuli esiin tarve toteuttaa arcade-tyylisille peleille suunnattu oma fysiikkamoottori, joka siis suorittaisi vain yksinkertaisimpia fysiikkamoottorin toimintoja. Ideana olisi saada siitä mahdollisimman kevytrakenteinen ja nopea. Fysiikan tarkastelussa jätettäisiin huomiotta kitkat, vierintävastukset ja kimmoisuudet. Pohjana tällaiselle fysiikkamoottorille voisi olla jonkin yksinkertaisen tasohyppelypelin fysiikan toteutus [7, sivut 291–396]. Haasteena tämänkaltaisen fysiikkamoottorin toteutuksessa on sen käytön rajaaminen siten, ettei sitä lähdettäisi vähääkään laajentamaan. Tällaisen itse toteutetun fysiikkamoottorin käyttöönoton yhteydessä olisi siis syytä määritellä tarkkaan se, milloin tulee siirtyä järeämmän fysiikkamoottorin käyttöön.

2.4.4 Fysiikkamoottoreiden tarkastelussa tehdyt huomiot

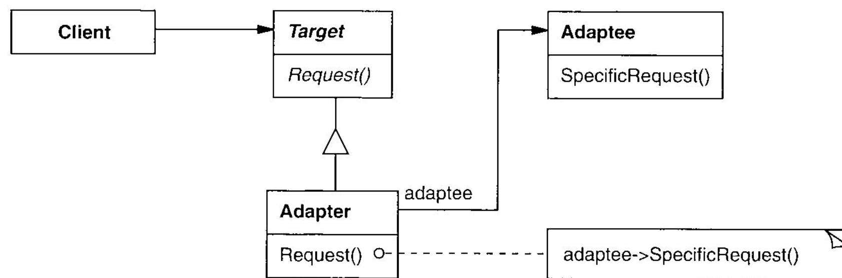
Fysiikkamoottoreiden suurehko poikkeavuus toisiinsa nähden oli ennalta arvattavissa oleva tulos. Tämä johtunee siitä, että fysiikkamoottoreiden rakennetta ei ole mitenkään yleisesti standardisoitu. Tämä on toisaalta hyvä asia, sillä fysiikkamoottoreihin kohdistuvat vaatimukset ovat vaikeasti ennalta arvattavissa, ja siksi onkin syytä olla saatavilla erityyppisesti toteutettuja fysiikkamoottoreita. Suurehkosta poikkeavuudestaan huolimatta fysiikkamoottoreita yhdistää se, että kaikilla niistä on fysiikkaobjekti, jolle annetaan erilaisia ominaisuuksia, kuten massa, kitkakerroin, hitausmomentti sekä voidaan asettaa objekti erilaisiin tiloihin. Fysiikkaobjekti oli nimeltään lähes poikkeuksetta joko `Body` tai `Rigidbody`, joista jälkimmäinen tarkoittaa suomeksi siis jäykkää objektia. Yksi adapterikirjaston perustehtävistä on kyseisen fysiikkaobjektin sovittaminen Jypeli-kirjaston ja fysiikkamoottorin välille.

3 Adapterikirjaston toteutus

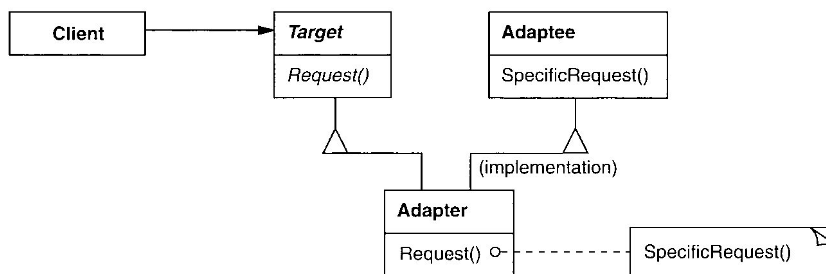
Adapterikirjaston toteutuksen alkuvaiheissa oli selvää, ettei heti ensimmäisenä tavoitteena kannattanut yrittää suoraan fysiikkaobjektien liikuttelua. Jo pelkkä vanhan fysiikkamoottorin poisto toi kerralla niin monta virheilmoitusta, että operaation systemaattinen eteneminen oli vaikeaa. Tavoitteet oli jaettava siten, että ensimmäisenä saadaan kääntäjä tyytyväiseksi käännösaikaisten virheiden osalta. Seuraavaksi jäljitettiin ohjelman suorituksen aikaiset virheet, jotka olivat suurimmalta osin nollaviitevirheitä. Viimeisenä tavoitteena oli saada aikaiseksi jokin fysiikkaobjektin interaktio adapterikirjaston kautta. Sen jälkeen adapterikirjastoa laajennettiin toimimaan kaikkien niiden toimintojen osalta, joita aikaisemmasta fysiikkamoottorista otettiin käyttöön. Tässä luvussa esitellään adapterikirjaston periaate sekä sen toteuttaminen kohta kohdalta.

3.1 Adapterikirjaston periaate ja suunnittelumalli

Adapterikirjaston periaate on yksinkertaisimmillaan mahdollistaa kahden eri kirjaston keskenään toimiminen siten, ettei kummankaan lähdekoodiin tarvitse tehdä muutoksia. Edellisessä luvussa todettiin tarkasteltavien fysiikkamoottoreiden käyttävän XNA 4.0 kehystä, sekä niissä olevan yhtenevänen jäykkä `Body`-objekti. Tässä tutkielmassa tarkastellaan siis vain jäykkiä fysiikkaobjekteja. Adapterikirjasto pyrkii olemaan mahdollisimman kevyt ja helposti muokattava. Huomioitavaa on se, että varsinaisia rajapintaluokkia on käytetty hyvin vähän. Syynä tähän on Jypeli-kirjaston luokat, jotka perivät jotain aikaisemmin käytetyn fysiikkamoottorin luokista. Adapterikirjaston ensimmäisessä versiossa on pyritty pitämään muutokset Jypeli-kirjastoon mahdollisimman vähäisinä.



Kuva 3.1: Oliosovitin käyttää olioiden koostamista [6, sivu 141].



Kuva 3.2: Luokkasovitin käyttää rajapintojen yhteensovittamisessa moniperintää [6, sivu 141].

Adapterin suunnittelumallissa voidaa soveltaa luokkasovitinta 3.2 tai oliosovitinta 3.1 Oliosovittimessa ovat osallistujat `Target`, `Client`, `Adaptee` ja `Adapter`. `Target` määrittelee `Clientin` käyttämän sovelluskohtaisen rajapinnan. `Client` taas käyttää `Target`-luokan mukaista rajapintaa. `Adaptee` määrittelee olemassa olevan rajapinnan, joka tarvitsee sovitusta. `Adapter` sovittaa `Adaptee`-luokan rajapinnan `Target`-luokan rajapintaan [6, sivu 141]. Käytetty oliosovitinmalli on tarkoitettu käytettäväksi pelkästään olioiden kanssa. Jypeli-kirjaston ja fysiikkamoottorin välille sovitinta sovitettaessa täytyi ottaa huomioon se, että tavallisten luokkien lisäksi sovitettavana on staattisia luokkia sekä `struct`-tyyppisiä luokkia eli tietueita. Tällaisissa tapauksissa on järkevintä soveltaa joko luokkasovitinta tai tehdä jokin kompromissiratkaisu aiemmin mainittujen sovitinmallien välillä.

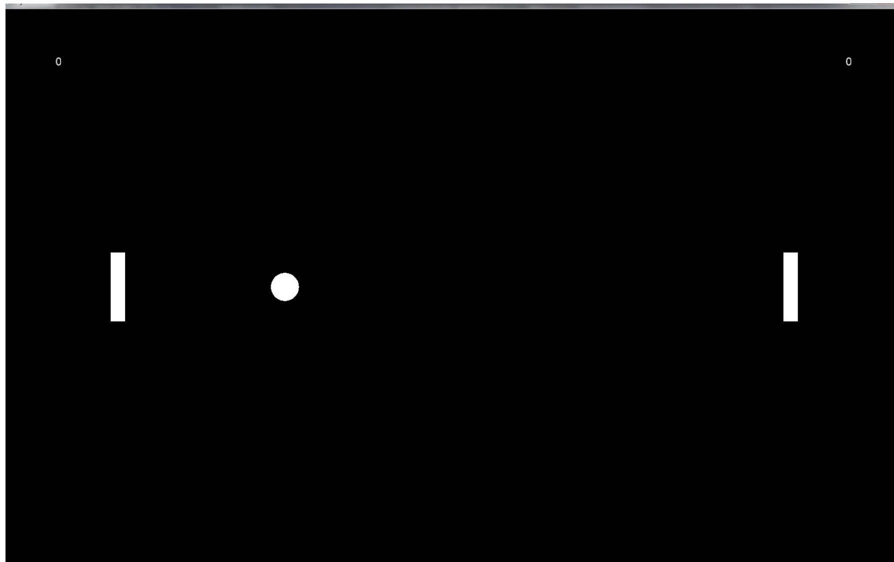
3.2 Testaus peleillä

Adapterikirjaston tehtävä on sovittaa fysiikkamoottori Jypeli-kirjaston avulla toteutettuihin peleihin, ja näin ollen on luontevaa testata adapterikirjastoa erilaisilla fy-

siikkapeleillä. Testitapauksina käytetään jo valmiiksi toteutettuja esimerkkipelejä, jotka ovat optimoituja toimimaan aikaisemmin käytetyn fysiikkamoottorin kanssa. Tämän vuoksi peleiltä ei voi odottaa identtistä käyttäytymistä eri fysiikkamoottoreiden kanssa. Testauspelit ovat valikoitu siten, että ne kattaisivat fysiikkamoottorilta vaadittuja ominaisuuksien kannalta mahdollisimman laajan alueen.

3.2.1 Objektin törmäyksen ja törmäyskulman testaaminen

Tietokone- ja videopelien historiassa yksi melko tunnettu julkaisu on ollut Pong. Kaikessa yksinkertaisuudessaan se on silti ollut innoittajana monelle pelikehittäjälle. Pelissä on kaksi pelaajaa, ja pelin tarkoituksena on saada pallo vastapelaajan mailan taakse. Mailat sijaitsevat kuvaruudun vasemmassa ja oikeassa laidassa, ja niitä voi liikuttaa pystysuunnassa. Pelin alkaessa pallolle annetaan alkuvauhti, ja tämän jälkeen nopeuteen voi vaikuttaa vain pallon iskeytymähetki mailan reunaan sen ollessa pallon suuntaisessa liikkeessä. Pong-pelillä voi hyvin testata pallon kimpoamiskulmia törmäyksen jälkeen, sekä tarkastella liikemäärän säilymistä liikemäärän säilymislain mukaisesti. Jypeli-kirjaston avulla tehdyssä Pong-versiossa on pistelaskurit, jotka kasvavat yhdellä pallon osuttua vastapelaajan takalaitaan. Maksimipistemäärän voi asettaa pelin alustustoimien yhteydessä, ja halutessaan voi myös vaikuttaa pallon muotoon ja sen fysiikan ominaisuuksiin, kuten esimerkiksi kitkaan, kimmoisuuteen ja massaan.

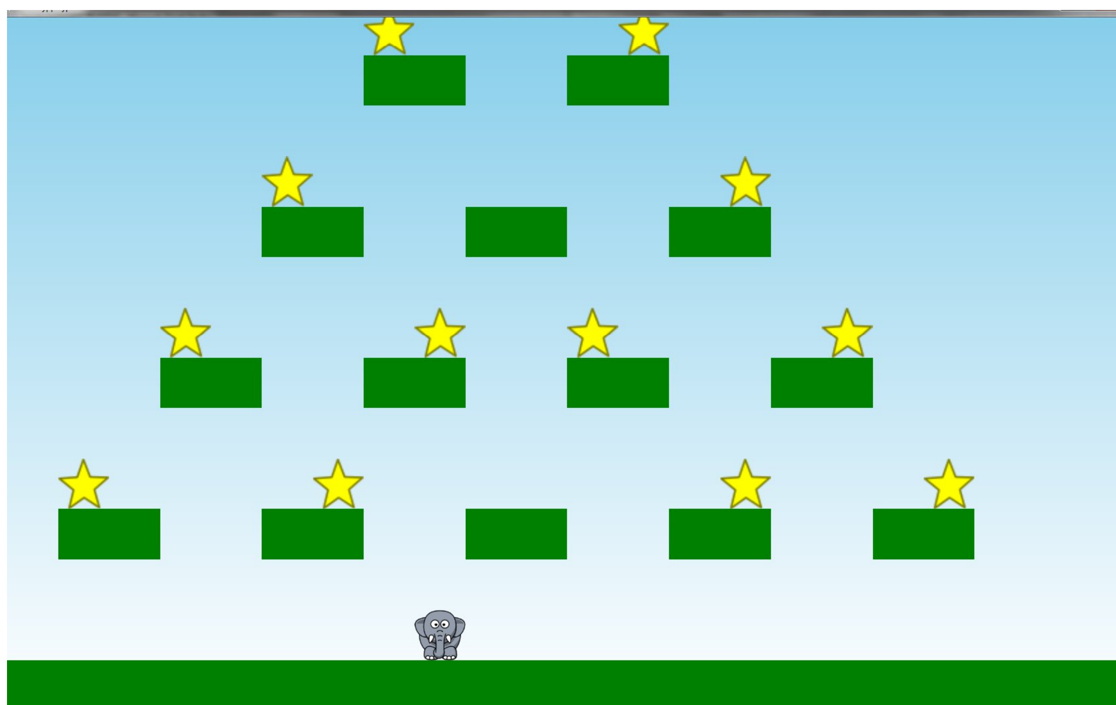


Kuva 3.3: Pong-peli on ehkä yksi maailman tunnetuimmista tietokonepeleistä.

Pong-peli on kehitetty Jypeli-kirjaston käytön opetustarkoitukseen. Se on nopea koodata malliesimerkin pohjalta. Näin ollen se ei tarjoa järin suurta haastetta testimielessä, mutta mikäli tämän pelin perustoiminnot eivät mene adapterista läpi, ei jatkokehittely ole enään järkevää. Staattisina objekteina ovat pelikentän rajat, ja dynaamisina objekteina ovat mailat ja pallo. Pelissä otetaan kiinni törmäystapahtuma, joka tapahtuu pallon osuessa jompaan kumpaan sivurajaan.

3.2.2 Objektiin vaikuttavien voimien manipuloinnin testaaminen

80-luvulla pelikonsolit toivat yksinkertaisten arcade-pelien rinnalle tasohyppelypelit. Yksiä tunnetuimpia näistä peleistä on varmasti Nintendon lanseeraama Super Mario Bros -pelisarja. Tasohyppelypeleille tyypillistä on niiden fysiikan lakeja uhmaavat iskunkestävät pelihahmot, sekä ilmassa leijuvat objektit, joiden keräämisestä pelaajaa palkitaan pisteillä. Tasohyppelypelien maailmat ovat yleensä laajoja, ja saattavat kukin sisältää useita tasoja. Joissain tasohyppelypelissä pelihahmo saa käytettäväksi aseita vastustajiensa eliminoimiseksi, tai kyvyn murskata vastustajansa hyppäämällä tämän päälle. Tasohyppelypelillä voi testata fysiikkamoottorin käyttöä Pong-peliä hieman vaativammassa ympäristössä.

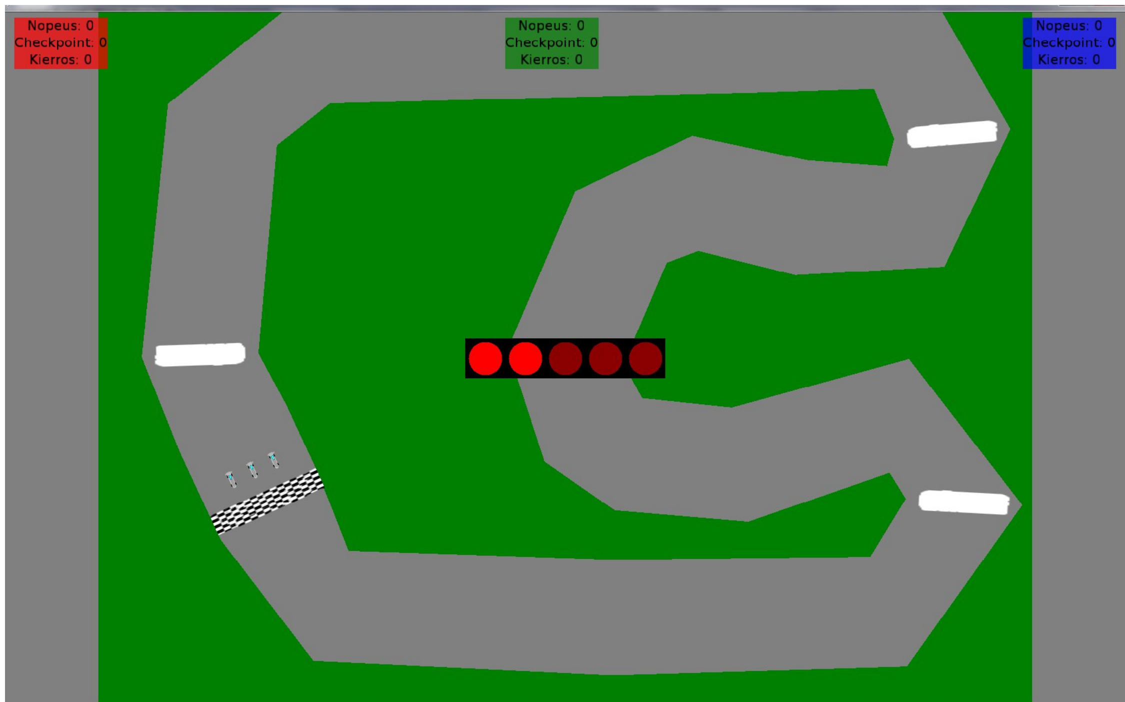


Kuva 3.4: Tasohyppelypeli on suosittu peligenre.

Tasohyppelypelissä adapterin täytyy välittää objektien paikkatiedot Jypeli-kirjastolle, ja viedä fysiikkamoottorille uudet nopeus- ja voimavektorit. Esimerkiksi hahmon hyppääminen saadaan aikaan voimaimpulssilla, joka antaa fysiikkaonjektille hetkellisen pystysuoran nopeuden. Tasohyppelypelin `PlatformCharacter`-objekti haastaa adapterin törmäystarkastelujen suhteen, sillä se sisältää varsinaisen peliobjektin lisäksi `CollisionHelper`-fysiikkaobjekteja, joiden on tarkoitus toimia ikäänkuin törmäysantureina.

3.2.3 Eri voimien yhteisvaikutuksen hallinnan testaaminen

Fysiikkapelien historiassa hyvin pian tasohyppelypelien jälkeen tulivat autopelit. Pääasiallinen perspektiivi on joko ylhäältäpäin kuvattu, tai ajoneuvon takaapäin kuvattu kolmiulotteinen perspektiivi. Fysiikan mallinnuksen tasot vaihtelevat tämän tyyppisissä peleissä laidasta laitaan. Tässä testissä käytetyssä esimerkissä autot kiertävät kilparataa, ja voittaja on se, joka on kierroksen jälkeen ensimmäisenä maalissa. Autot voivat törmäillä toisiinsa, ja radalta ulos ajaminen vähentää auton nopeutta. Fysiikan testaamisen näkökulmasta tämänlaisella pelillä voi todeta sen, että käyttäytyykö auto kurveissa mielekkäällä tavalla ja ovatko törmäyskulmat kohdallaan, sekä näiden kaikkien yhteisvaikutuksen.

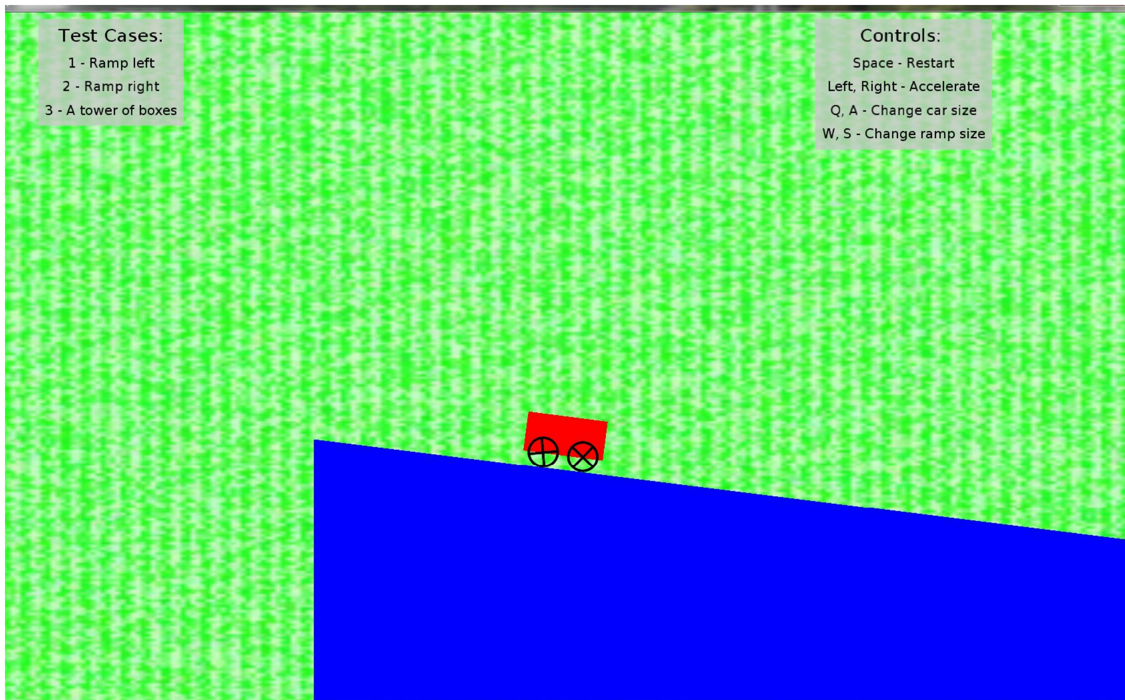


Kuva 3.5: Lintuperspektiivistä kuvatty autopeli.

Ylhäältä päin kuvatussa kilpa-autopelissä adapteria kuormitetaan hyvin pitkälti samojen ominaisuuksien osalta, kuin mitä tashyppelytestipelissäkin. Vakiona säilyvän painovoiman sijaan autopelissä vallitsevat alati vaihtelevat keskipakois- ja kitkavoimat. Törmäystapahtumia ei tässä esimerkissä rekisteröidä.

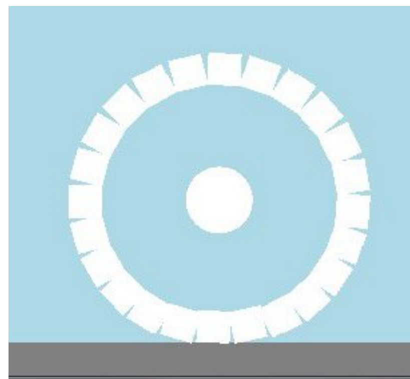
3.2.4 Fysiikkaobjektien välisten akseliliitosten testaaminen

Pidemmälle kehittyneissä fysiikkapohjaisissa peleissä saatetaan pelihahmo koota useista komponenteista luomaan todentuntuisuutta. Sivultapäin kuvatuissa autopeleissä ja vapaalla viivalla piirretyssä maastossa saadaan hienoja pelikokemuksia määrittelemällä renkaat omiksi itsenäisiksi fysiikan objekteiksi. Tässä tutkielmassa käytetyssä esimerkkipelissä ei yksityiskohdilla hienostella. Esimerkin tarkoituksena on osoittaa, että akseliliitoksella liitetty objekti pysyy oikealla paikallaan ja käyttäytyy odotetusti. Tässä tapauksessa odotetaan aidontuntuista ajoneuvon kiihdytystuntumaa.



Kuva 3.6: Sivulta päin kuvattu autopeli.

Adapterikirjaston on luotava akseliliitokset, sekä annettava vetävälle pyörälle vääntömomentti. Fysiikkaobjekteista koostettu ajoneuvo saa liike-energiansa rekaiden kehänopeuden sekä niiden ja maanpinnan välisen kitkan vaikutuksesta.

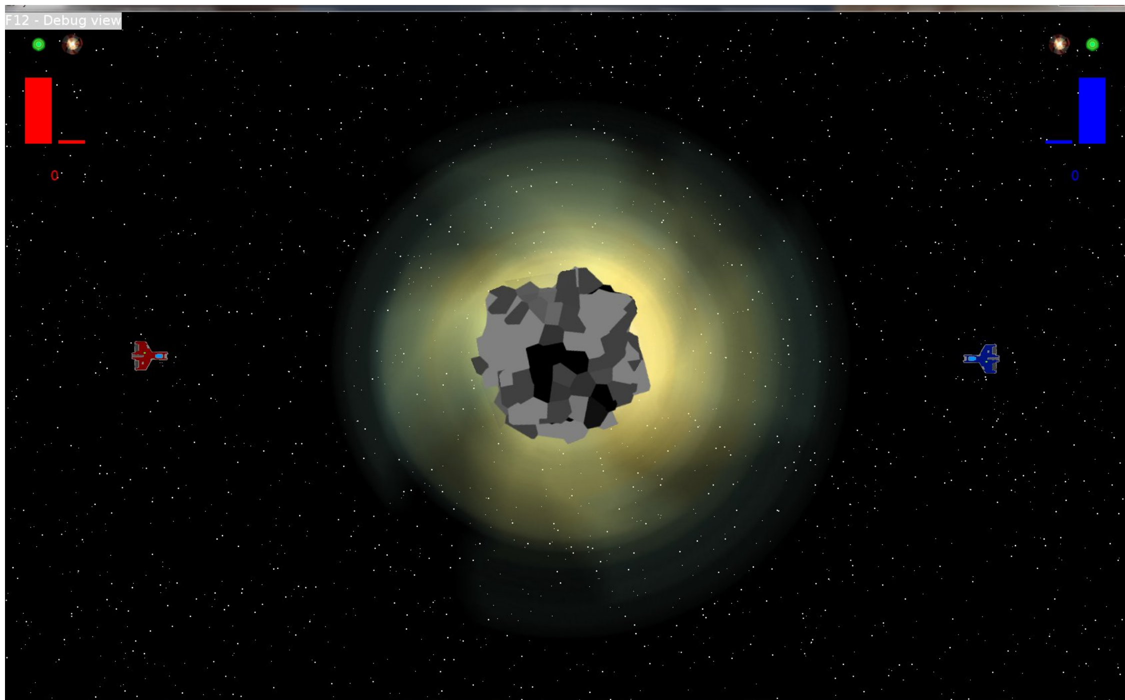


Kuva 3.7: Telaketjurenkaalla voi testata ei-jäykkien kappaleiden törmäyksiä.

Fysiikkaobjektien välisiä akseliliitoksia voi käyttää autopelien lisäksi monessa muussakin yhteydessä. Toisessa akseliliitoksiin liittyvässä testipelissä ohjataan telaketjumaista rengasrakennetta. Akseliliitokset ovat joustavia, ja siten saadaankin aikaisek-si törmäyksen löysä vesi-ilmapallomainen efekti. Joustavien akseliliitosten vastaavuudesta eri fysiikkamoottoreissa ei ole mitään takeita. Mikäli fysiikkamoottori ei tarjoa joustavia akseliliitoksia, täytyy ne silloin määritellä jäykiksi. Se saattaa kuitenkin aiheuttaa fysiikan mallinnuksessa täysin odottamattomia ilmiöitä.

3.2.5 Projektiilien testaaminen

Arcade-pelieistä tunnetun asteroids-pelin klooni löytyy myös Jypeli-projektiryhmän esimerkkipeleistä. Siinä kaksi avaruusalusta mittelevät asteroidin herruudesta. Avaruusaluksien liikuttelu toimii samalla periaatteella, kuin mitä aikaisemmin mainittu ylhäältäpäin kuvattu autopeli. Lisänä on ammukset, jotka ovat itsessään myös fysiikkaobjekteja. Tässä pelissä ei ole vaikuttavaa painovoimaa, ei edes kuvaruudun keskellä olevan asteroidin suhteen.



Kuva 3.8: Eräs versio Asteroids-pelistä.

Adapterikirjastossa on pyrittävä ottamaan huomioon fysiikkamoottorin projektiileille tarjoama erityinen fysiikanmallinnus. Ammuskelupeleistä tankkipeli koettelee edellisen lisäksi projektiilien fysiikkaominaisuuksia, sillä tässä testipelissä on painovoima, joka vaikuttaa myös ammuksiin. Sen lisäksi pelimaailma alkaa vähitellen täyttymään vastustajista, jotka ovat myös fysiikkaobjekteja. Peli on selvitymistyylinen, eli vastuksia lisätään pelimaailmaan niin kauan, kunnes pelaaja saa riittävän monta osumaa itseensä.

Kuva 3.9: Tankkipelissä koetellaan selviytymistaitoja.

Pelissä jatkuvasti kasvava fysiikkaobjektien määrä kuormittaa fysiikkamoottoria. Mikäli fysiikkamoottori ei tarjoa omaa fysiikanmallinnusta pienille partikkeleille, niin ne joudutaan käsittelemään projektiilien sijaan normaaleina fysiikkaobjekteina. Mikäli fysiikkamoottorin ominaisuudet tämän sallivat ilman fysiikan mallinnuksen laadun heikkenemistä, voidaan projektiilit jättää huomioimatta erikoistapauksina.

3.3 Kokeilu- ja kehitysprojektit

Adapterikirjaston toteuttaminen tehtiin vaiheissa, jotka nimettiin kokeilu- ja kehitysprojekteiksi. Jokaista vaihetta ja sen raportointia varten on varattu viikko työai-

kaa. Kokeiluprojekteissa pääpaino on Jypeli-kirjaston ymmärryksen kartuttamisessa erilaisin kokeiluin. Kokeiluprojekti päättyi joko kokeilun hyväksymiseen tai hylkäämiseen. Hylkäämisen perusteena oli joko työmäärän kasvaminen tarpeettoman suureksi tavoitteen tärkeyteen nähden tai asetettujen tavoitteiden arvoitu saavuttamatta jääminen. Hyväksymisen perusteena oli tavoitteen saavuttaminen, ja hyväksytty kokeilu siirrettiin uudeksi kokeiluksi uudella kokeilunumerolla ja tavoitteella. Uudeksi fysiikkamoottoriksi kokeiluprojekteihin valittiin Farseer Physics.

Kehitysprojekteissa nimensä mukaisesti tähdättiin adapterikirjaston kehittelyyn Jypeli-kirjastosta saadun tietämyksen turvin. Aikaisempien kokeiluprojektien tulosten perusteella adapterikirjaston kehittely oli aiheellista. Kehitysprojekteissa edettiin vaihe vaiheelta alkaen yksinkertaisen ennalta määritellyn Pong-pelin toteuttamisesta aina tasohyppelypeliin, ja sitä kautta muihin pelityyppeihin. Tavoitteena kehitysprojekteilla oli saada adapterikirjaston valmiusaste niin pitkälle kuin vain mahdollista, aina käyttöönottoon asti.

3.3.1 Kokeiluprojekti 1

Ensimmäisessä kokeiluprojektissa oli tarkoitus vaihtaa fysiikkamoottori valmiiseen esimerkkipeliprojektiin siten, että ensiksi poistetaan konkreettisesti jo käytössä oleva fysiikkamoottori ja liitetään uusi käyttöönotettava fysiikkamoottori tilalle. Sen jälkeen oli tarkoitus tehdä Jypeli-kirjastoon tarvittavat muutokset muuttamalla korvattavat rivit kommenttiriveiksi, ja lisäämällä niiden perään tunnisteen "Physics2D". Uudet lisätyt rivit merkittiin tunnisteella "Farseer".

Tässä kokeilussa saatin hyvin selville se, että minkälaiset kytkökset vallitsevat Jypeli-kirjaston ja fysiikkamoottorin välillä. Virheitä korjatessa niitä paljastuu jatkuvasti enemmän ja enemmän, sillä kääntäjä ei anna heti suoraan kaikkien virheiden tarkkaa määrää. Kokeiluprojekti hylätään, sillä eteneminen ilman välitestauksia ei anna riittäviä takeita kokeilun onnistumisesta.

3.3.2 Kokeiluprojekti 2

Toisessa kokeiluprojektissa aloitettiin luomaan adapterikirjastoa niin sanotusti tyhjältä pöydältä. Valmiista esimerkkipeliprojektista poistettiin siinä valmiina oletuksena tullut Jypeli-kirjasto, ja luotiin uusi tyhjä Jypeli-projekti. Esimerkkipelin tyh-

jään Jypeli-kirjastoon tuotiin sitten luokkia oikeasta Jypeli-kirjastosta yksitellen rakentaen samaan aikaan adapterikirjastoa sen mukaan, mitä Jypeli-kirjastosta tuodut luokat fysiikkamoottoria tarvitsivat.

Tässä kokeilussa käännösaikaisten virheiden määrä pysyi hallittuna. Koska esimerkkipelissä käytettiin vaillinaista Jypeli-kirjastoa, ei lähdekoodia pystynyt saamaan kääntäjästä virheittä läpi. Lisättyjen rivien jälkeen olisi kuitenkin syytä pystyä kääntämään ja testaamaan koodia. Kokeilu päätyi hylkäykseen vaikean testattavuutensa ja sitä myötä heikon toteutuvuutensa vuoksi.

3.3.3 Kokeiluprojekti 3

Kolmannessa kokeiluprojektissa asetettiin fysiikkamoottorit rinnakkain, eli jätettiin poistamatta vanha fysiikkamoottori uuden lisäämisen jälkeen. Tällä tavoin aloitettiin siirtämään Jypeli-kirjaston kytköksiä yksi kerrallaan vanhasta fysiikkamoottorista adapterikirjastoon. Jokaista Jypeli-kirjaston vanhasta fysiikkamoottorista tarvitsemaa luokkaa vastaamaan tehtiin vastaavanniminen luokka adapterikirjastoon. Tässä kokeilussa ei lisätty uutta fysiikkamoottoria, vaan kääntäjä pidettiin tyytyväisenä adapterikirjastoon lisättyjen luokkien niin sanottujen dummy-funktioiden avulla. Kyseinen funktio ei siis sisällä mitään toiminnallisuutta, ja palauttaa vain arvon 0, false tai nollaviitteen.

Physics2DDotNet-fysiikkamoottorin korvaaminen adapterikirjastolla saatiin siihen pisteeseen, että lähdekoodi menee kääntäjästä läpi. Tämä voitiin todeta konkreettisesti poistamalla vanha fysiikkamoottori kokonaan projektikansiosta. Poistossa ilmeni kymmenkunta uutta virhettä, jotka olivat kuitenkin triviaalisti korjattavissa. Ohjelman käynnistäminen sai aikaan odotetusti nollaviitevirheilmoituksia. Tavoitteena oli kuitenkin vain saada lähdekoodi kääntäjästä virheittä läpi, ja näin ollen kokeiluprojekti siirrettiin hyväksyttynä seuraavaan vaiheeseen.

3.3.4 Kokeiluprojekti 4

Neljännes kokeiluprojekti alkoi lähdekoodin siistimisellä ja tarkistamisella. Tarkistuksessa todettiin, että adapterikirjastossa olevat luokat vastasivat tyypiltään sitä Physics2DDotNet-fysiikkamoottorin luokkaa, jota adapterikirjastossa olevan luokan oli tarkoitus jäljitellä. Kaikki metodit varmistettiin olevan määritelty julkisiksi,

ja jokainen property-ominaisuus automaattisella get-set:llä. Tässä kokeiluprojektissa poistettiin myös Physics2DDotNet-fysiikkamoottorin käyttämä AdvanceMath-kirjasto samoin menetelmin, kuin mitä fysiikkamoottorikin. Jotkin Jypeli-kirjaston käyttämät triviaalit laskutoimitukset korvattiin System.Math-standardikirjaston vastaavilla metodeilla.

Nollaviitevirheiden korjaus tehtiin viemällä nollaus niin sanotusti ylemmälle tasolle. Esimerkiksi vektorin palauttavaan metodiin vaihdettiin null-viitteen palauttamisen sijaan nollavektorin palauttaminen. AdvanceMath-kirjaston adapteriluokat toteutettiin muusta poiketen metodien edellyttämillä laskutoimituksilla. Edellä mainittujen toimenpiteiden, ja muutaman virheenkorjauksen jälkeen esimerkkipeli käynnistyi piirtäen peliobjektit ruudulle niiden oikeille paikoilleen. Kokeiluprojekti siirtyi hyväksyttynä seuraavaan vaiheeseen.

3.3.5 Kokeiluprojekti 5

Viidennessä kokeiluprojetissa oli tarkoitus saada ensimmäinen fysiikkaobjektin interaktio, eli viedä fysiikkamoottorille fysiikkaolio sekä antaa sille voimavektori ja aikaintervalli. Paluuarvona odotettiin uutta sijaintia koordinaatistossa. Tätä kokeiluprojektia varten toteutettiin staattinen monitorointiluokka `Adapter` sekä `Windows Forms` -tyyppinen monitori-ikkuna fysiikkaobjektin sijainnin ja muiden ominaisuuksien tarkkailuun. Jypeli-kirjastosta tuleva dynaaminen fysiikkaobjekti saatiin kiinni sille annetun massa-arvon avulla, ja sen perusteella oli helppo luoda uuteen fysiikkamoottoriin vastaava fysiikkaobjekti.

Fysiikkaobjektin liikuttelu saatiin onnistumaan sivusuuntaisien voimavektoreidena ja painovoimavektorin avulla. Näiden avulla pystyttiin toteamaan aikaisemman fysiikkamoottorin käytön aikana ilmaantuneiden ongelmien poistuminen. Hyppytoiminnon ja muun edistyneemmän toiminnallisuuden käyttöönotto jätettiin myöhempiin vaiheisiin. Kokeiluprojekti siirtyy hyväksyttynä eteenpäin seuraavaan vaiheeseen, eli kehitysprojekteihin.

3.3.6 Kehitysprojekti 1

Ensimmäisessä kehitysprojektissa tavoitteena oli Pong-pelin fysiikan toteuttaminen. Paikka- ja nopeusvektorien lisäksi tässä projektissa täytyi ottaa huomioon myös kul-

ma. Koska Farseer Physics käyttää fysiikkaobjektien luokittelussa staattista, dynaamista ja kinemaattista määritystä, luokiteltiin Pong-pelin fysiikkaobjektit siten, että pallo oli dynaaminen, mailat kinemaattistia ja kentän rajat staattisia objekteja. Myöhemmissä kokeiluissa ilmeni, että staattiset objektit voidaan luokitella myös kinemaattisiksi. Kaikkien objektien määrittäminen dynaamisiksi aikaisemman fysiikkamoottorin tapaan ei tuottanut toivottuja tuloksia tässä vaiheessa.

Tämän kehitysprojektin myötä adapterikirjaston avulla pystyttiin tarkastelemaan fysiikkaobjektien käyttäytymistä myös pyörimisen suhteen. Pyörimättömyys tehtiin keinotekoisesti samalla tavalla kuin mitä aikaisemmassa fysiikkamoottorissa, eli määrittelemällä inertialle ääretön arvo. Ympyrän muotoisten kappaleiden lisäksi myös kantikkaat muodot pystyttiin tarkastelemaan kimpoamisien ja pyörähdysten suhteen. Pikaisessa tarkastelussa liike-energiaa ei kadonnut pelin aikana, ja näin ollen ensimmäinen kehitysprojekti päättyi onnistuneena.

3.3.7 Kehitysprojekti 2

Toisessa kehitysprojektissa oli tarkoitus ottaa Jypelissä oleva `PlatformCharacter`-luokka käyttöön. Kyseinen luokka sisältää tasohyppelypelin pelihahmoon liittyviä ominaisuuksia fysiikkaobjektin lisäksi. Tällaisia ovat muun muassa aiemmin mainitut `CollisionHelper`-fysiikkaobjektit sekä muut attribuuteilla ohjatut tasohyppelyhahmon ominaisuudet. Tavoitteena oli toteuttaa Jypeli-kirjaston malliprojektin mukainen tasohyppelypeli siten, että fysiikkahahmo tunnistaa sen objektin, minkä päällä milloinkin on, ja näin ollen mahdollistaa siihen ehdollistetun hyppäystoiminnan toteuttamisen.

Lopputuloksena tämän projektin jälkeen saatiin adapterikirjasto, joka pystyy välittämään Jypeli-kirjaston ja fysiikkamoottorin välillä oleellisimmat toiminnot. Adapterikirjaston kehittäminen staattisen `Adapteri`-luokan avulla ei tästä eteenpäin ole enää mielekästä, sillä Pong- ja tasohyppelypelin adapterikirjastot eivät ole identtisiä, eikä näin ollen ole keskenään yhteensopivia. Nopeusvektorien käsittely aiheutti haasteita etenkin Pong-pelissä, jossa pallon aloitusnopeus annetaan jo ennen varsinaisen fysiikkaobjektin lisäystä fysiikkamoottoriin. Tämän kehitysprojektin tavoitteet kuitenkin saavutettiin onnistuneesti.

3.3.8 Kehitysprojekti 3

Kolmannen kehitysprojektin päätarkoitus oli viedä adapterikirjasto olioympäristöön staattisen `Adapteri`-luokan sijaan. Tämän toteuttaminen edellytti joidenkin luokkien siirtämisen Jypeli-kirjastosta adapterikirjastoon. Turhiksi jääneet luokat poistettiin sekä adapterikirjastosta, että Jypeli-kirjastosta. Uusina käyttöönotettavina ominaisuuksina olivat fysiikkaobjektien väliset akseliliitokset, projektiilit sekä fysiikkaobjektin luominen bittikarttapohjaisesta tekstuurista. Testitapauksiksi valittiin yhden sijaan useita eri esimerkkipelejä.

Oliomalliseen adapterikirjastoon siirtyminen sujui suhteellisen vaivattomasti, ja tavoitteena olleet ominaisuudet toteutuivat riittävän yksityiskohtaisesti. Akseliliitosten tekemisessä pitäydyttiin toistaiseksi jäykissä akseliliitoksissa, joissa ei voi määrittää liitoksen pehmeyttä millään lukuarvolla. Turhien luokkien lisäksi poistettiin `PlatformCharacter2`-luokka, sillä sen käyttöönottamiseen uuden fysiikkamoottorin kanssa ei ole riittävästi perusteita. Esimerkkipelin suorituksenaikainen akseliliitosten poistaminen fysiikkamoottorista aiheutti toistaiseksi joitain virhetilanteita.

3.3.9 Kehitysprojekti 4

Neljännen kehitysprojektin tavoitteena oli luoda helposti testattava ja käyttöönotettava adapterikirjasto. Käytännössä tämä tarkoitti sitä, että käytettävä fysiikkamoottori voitiin vaihtaa lähdekoodissa kehitysvaiheessa yhden koodirivin avulla. Tavoitteena oli myös vähentää if-lauseiden ja fysiikkamoottorin tehdasmetodien kutsujen määrää lähdekoodissa, ja siten parantaa adapterikirjaston ylläpidettävyyttä. Tässä kehitysprojektissa testauksen ja virheiden korjauksien osuus tuli olemaan aiempaa merkittävästi suuremmassa roolissa.

Kehityksen aikainen fysiikkamoottorin vaihto vaatisi koko Jypeli-kirjastolta pitkälle vietyä modulaarisuutta. Käytännössä kaikki luokat, jotka käyttivät jotain fysiikkamoottorin resurssia, siirrettiin adapterikansion alle. Sen lisäksi tiedostojen nimiin lisättiin etuliitteeksi niiden käyttämän fysiikkamoottorin tunniste. Tähän mennessä selvittämättömästä syystä `Body`-olioiden luonti ja lisääminen Farseer Physics -fysiikkamoottoriin ei toistaiseksi onnistu kuin pelkästään tehdasmetodien avulla.

3.3.10 Kehitysprojekti 5

Viimeisessä kehitysprojektissa ei enää uusia ominaisuuksia oteta käyttöön, vaan projekti keskittyy pääpainoisesti testaamiseen ja viimeistelyyn. Viimeisenä vaiheena poistetaan fysiikkamoottoria käyttävistä luokista fysiikkamoottoria kuvaavat etuliitteet pois. Testaamisessa käytetään aikaisempien testipelien lisäksi peliohjelmointikurssilaisten tuottamia pelejä. Lopullinen adapterikirjasto ei ole täydellinen, mutta kykenee hoitamaan yksinkertaisimpien pelien fysiikan mallinnuksen uuden fysiikkamoottorin avulla.

Fysiikkamoottoria kuvaavien etuliitteiden poisto luokkien nimien edestä onnistui ilman päällekkäisyysongelmia. Peliohjelmointikurssilaisten pelejä testatessa tuli ilmi puutteita sellaisen pelin fysiikan mallintamisessa, jossa pelaajat liikkuvat ylhäältäpäin kuvatussa maailmassa. Nämä pelit olivat poikkeuksellisesti toteutettu tavallisen `PhysicsGame`-luokan avulla Jypelissä olevan `TopDownPhysicsGame`-luokan sijaan. Nämä puutteet tullaan korjaamaan adapterikirjaston mahdollisessa jatkokehittelyssä ja käyttöönoton yhteydessä.

3.4 Havaitut ongelmakohdat ja ratkaisut perusteluineen

Adapterikirjaston kehittämisen aikana tuli eteen useita ongelmakohtia, joista osa johtui fysiikkamoottorin odottamattomista ominaisuuksista ja loput tekijän omasta huolimattomuudesta. Tässä kappaleessa käydään läpi muutamia eteen tulleita ongelmakohtia, joita voi esiintyä jatkossa muita fysiikkamoottoreita adapterikirjastoon sovitettaessa.

Jo heti alussa Farseer Physics -fysiikkamoottoria tutkittaessa tuli ilmi sen käyttämät liukuluvut, jotka olivat float-tyyppisiä. Jypeli-kirjastossa liukulukuja käsitellään double-tyyppisinä. Näiden oleellisin ero on se, että float-tyyppi on 32-bittinen ja double 64-bittinen. Käytännössä tämä tarkoittaa double-tyyppisten liukulukujen konvertoimista float-tyyppisiksi adapterikirjastossa. Hiuksenhienoilla tarkkuuksilla pelatessa saattaisi jotakin epätäsmällisyyksiä ilmetä, mutta mikäli pystytään huolehtimaan siitä, että virheet eivät rupea kertautumaan, tätä kokoluokkaa olevat poikkeavuudet eivät vaikuta pelattavuuteen.

Farseer Physics -fysiikkamoottorin valinnassa oli keskustelupalstojen suositusten lisäksi luokkarakenteen jonkinasteinen samankaltaisuus Physics2DDotNet-kirjastoon verrattuna. Vaikka eri kirjastoissa samannimisiä luokkia olisikin, ei niiden voida olettaa olevan yhteneväisiä. Farseer Physics pyrkii selvästi olemaan helppokäyttöinen ja selkeä fysiikkamoottori. Siinä missä Physics2DDotNet tarjoaa lähtökitkan ja liikekitkan, Farseer tarjoaa pelkästään kitkan. Toteutetussa adapterikirjastossa viedään ainoastaan liikekitka Farseer Physics -kirjastolle. Vaikutukset pelattavuuteen ovat hiuksenhienot.

Physics2DDotNet-kirjastosta poiketen Farseer Physics jakaa fysiikkaobjektit kolmeen kategoriaan, eli staattiseen, dynaamiseen ja kinemaattiseen. Jypeli-kirjastosta tulevien fysiikkaobjektien jaottelu näihin kategorioihin ei ole täysin triviaalia. Toistaiseksi kategorisoimisessa on päädytty jakamaan objektit dynaamisiin ja kinemaattisiin fysiikkaobjekteihin. Testaukset muutamilla peleillä osoittivat kinemaattisen objektin täyttävän staattisen objektin kriteerit, mikäli sen nopeusvektoreihin ei kosketa. Pelkkien dynaamisten fysiikkaobjektien käyttö voi jatkossa tulla kysymykseen.

Adapterikirjaston testausten alkuvaiheessa ilmeni useita odottamattomia ilmiöitä. Pelin suoritusaikaisten virheiden jäljittäminen on varsin haastavaa, sillä peli täytyisi saada pysähtymään juuri virheen sattuessa. Peliohjelman suorittaminen debugtilassa rivi riviltä ei ole mielekästä, sillä pelin päivitysmetodi suoritetaan keskimäärin 60 kertaa sekunnissa. Esimerkiksi pitkään esillä ollut `PlatformCharacter`-fysiikkaobjektin kummallinen törmäily origossa johtui `CollisionHelper` -fysiikkaobjekteista, jotka paikoittuivat nollavektorin mukaisesti. Virhe korjaantui adapteriluokan parantelulla, mutta itse virheen jäljittäminen oli haastavaa.

Hitausmomentin määrittely kappaleelle tapahtuu yleensä samassa yhteydessä, kuin mitä kappaleen pelimaailmaan luominen. Jossakin pelitilanteessa halutaan ehkä vaikuttaa kappaleen käyttäytymiseen inertian muuttamisen avulla. Muutamassa testitapauksessa inertian muuttaminen kappaleen luomisen jälkeen sai aikaan ei toivottuja ilmiöitä, kuten esimerkiksi objektin putoamisen tason läpi. Toistaiseksi tämän ilmiön kanssa toimitaan siten, että adapterikirjasto ei vie Farseer Physics -fysiikkamoottorille inertian arvoa `PhysicsObject`-luokasta fysiikkaolion luomisen yhteydessä.

Adapterikirjaston rakenne ideaalisimmillaan jäljittelisi aikaisemman fysiikkamoottorin ympärille muodostunutta rakennetta. Farseer Physics -fysiikkamoottorissa siinä olevien esimerkkien perusteella suositeltu tapa on fysiikkaolion luonnin yhteydessä määrittää sille maailma. Physics2DDotNet-fysiikkamoottorissa sen sijaan fysiikkaobjekti liitetään maailmaan vasta myöhemmissä vaiheissa. Piirre ei itsellään aiheuta ongelmia, mutta fysiikkaobjektiluokan rakenteessa on jouduttu tekemään joitain rakenteellisia kompromissiratkaisuja sen suhteen, että saadaan fysiikkaobjektit luotua turvallisesti ja varmasti.

Kahden toisiinsa törmäävän objektin kimpoamisesta on ainakin kaksi eri koulukuntaa. Mikäli toinen kappaleista on täysin kimmoton ja toinen kimmoinen, tapahtuuko jomman kumman kappaleen kimpoamista ylipäänsä? Physics2DDotNet määrittelee kimpoamattoman kappaleen siten, että se imee molempien kappaleiden liike-energian. Farseer Physics taas suhtautuu kimmoisuuteen kappalekeskeisesti, eikä kappaleen kimmottomuus poista törmänneen kimmoisen kappaleen liike-energiaa. Tällaisissa tilanteissa täytyy pelin lähdekoodissa määritellä kimmoisuudet käytettävän fysiikkamoottorin mukaisesti.

Testattaessa suurilla objektien lukumäärällä ja yli tuhannen suuruisilla voimavektoreilla Farseer Physics -fysiikkamoottoria huomattiin, että pallot alkavat hiljalleen uppoamaan toistensa sisään. Tästä voidaan päätellä, että suurilla objektien määrällä ja voimilla mutta epätarkalla törmäyssuuntien määrittelyllä Physics2DDotNet -fysiikkamoottori selviytyy fysiikan mallinnuksesta paremmin, ja vastaavasti Farseer Physics toimii parhaiten pienemmillä objektien määrillä ja voimilla. Syynä pallojen toisiinsa uppoamiseen voi myös olla Jypeli-kirjaston aiheuttama kuormitus yhden ohjelmakierron aikana. Tämä ilmiö kuitenkin puoltaa adapterikirjaston toteuttamista siten, että kytkettyjen fysiikkamoottoreiden vaihto olisi sujuvaa.

Luokkien ja metodien nimeämisessä tulisi välttää samojen nimien käyttöä, kuin mitä projektissa käytettävissä kirjastoissa on jo ennestään. Tämä helpottaa huomattavasti lähdekoodin luettavuutta, ja vähentää oleellisesti ajatusvirheiden syntymistä. Esimerkiksi `Game`-niminen luokka voisi olla Jypeli-ympäristössä nimetty `JGame`-nimellä, tai jotenkin vastaavasti. Lähdekoodin rivien ja luokkien määrän kasvaessa nimiristiriidan esiintymisen todennäköisyys kasvaa. Ristiriitojen määrä vähenee, kun erillään toimivien kokonaisuuksien nimiavaruudet pidetään erillään.

4 Adapterikirjaston rakenne

Kuten jo teoriaosassa käsitellyssä olio- ja luokkasovittimen yhteydessä mainittiin, on tässä tutkielmassa toteutettavassa adapterikirjastossa tehtävä kompromisseja sen suhteen, miten paljon sovitinmallia voidaan jäljitellä. Tekijän kokemattomuuden vuoksi oli työssä edettävä hyvin maltillisesti pieniä kokonaisuuksia kerrallaan toteuttaen. Adapterikirjastosta syntyi kehitysaikainen staattinen versio, sekä toteutuskelpoinen oliomallinen versio. Adapterikirjasto ei itsessään sisällä muutaman pienen matemaattisen laskutoimituksen lisäksi mitään omaa toiminnallisuutta, vaan sen tarkoitus on muuntaa osapuolien välinen tiedonsiirto niille oikeaan muotoon.

4.1 Adapterikirjaston toimintaperiaate rajapintana

Kokeiluvaiheen staattisessa adapterikirjastossa tehtiin kaikista Jypeli-kirjaston käyttämistä Physics2DDotNet-kirjaston luokista dummy-luokat. Luokkien määritteleminen suoraan rajapinnoiksi interface-määrittelyllä olisi tullut kysymykseen, mikäli Jypeli-kirjastossa ei olisi ilmennyt aikaisemmasta fysiikkamoottorista perittyjä luokkia. Dummy-luokkien joukossa oli myös `struct`-tyyppisiä luokkia, eli tietueita. Jypeli-kirjaston ja adapterikirjaston välisen rajapinnan oleellisimmat luokat olivat Physics2DDotNet-kirjastosta otetut `PhysicsEngine`, `GravityField` sekä `PhysicsObject`-luokan käyttämä `Body`-luokka, sillä Jypeli-kirjaston `PhysicsGame`-luokassa on viitteet `PhysicsEngine`- ja `GravityField`-luokkiin, ja `PhysicsEngine` sisältää siihen lisätyt `Bodyt` listana. Kaikki dataliikenne hoidetaan kuitenkin vain yhden staattisen `Adapteri`-luokan kautta. Toteutusvaiheen oliomallisessa adapterikirjastossa käytetään samoja luokkia rajapintana, kuin staattisessa versiossa. Oliomallisessa versiossa staattisen luokan sijaan kytketään luokat toisiinsa siten, että esimerkiksi `PhysicsGame`-luokassa luodaan `PhysicsEngine`-luokalle vastine `World`.

Farseer Physics tarjoaa omana rajapintanaan `World`-luokan, jolle voi joko luonnin yhteydessä, tai jäljestä päin määrittää painovoimavektorin. Erillistä painovoimaluokkaa ei siis ole. Staattisessa toteutuksessa `World`-luokan ilmentymä luodaan

staattisen Adapteri-luokan attribuutiksi, ja GravityField-luokan ilmentymää luodessa viedään Adapteri-luokalle painovoimavektori, jossa se liitetään edelleen World-olioon. Voima-, nopeus-, ja impulssivektorit välitetään luomalla uudet vektorit Farseer Physics Body-oliolle. Physics2DDotNet ja FarseerPhysics Body-oliot sisältävät viitteet toisiinsa sen sijaan, että niistä olisi moniperinnällä koostettu yksi universaali Body-luokka. Toteutusperiaatteella aggregaation tai komposition avulla ei kokeiluvaiheessa ole juurikaan käytännön merkitystä. Toteutusvaiheen oliomallisessa versiossa rajapinta FarseerPhysics-fysiikkamoottorille päin ei käytännössä eroa staattisesta versiosta. Myöhemmissä kehitysvaiheissa on mahdollisuutena vähentää FarseerPhysics-fysiikamoottorin tehdasmetodien käyttöä lisäten siten parempaa lähdekoodin ylläpidettävyyttä.

4.2 Kokeiluvaiheen staattinen versio

Adapterikirjaston luokkarakenne on hyvin selkeä ja yksinkertainen. Kirjasto sisältää 33 dummy-luokkaa, jotka ovat kopioitu Physics2DDotNet-kirjastosta, mutta sisältävät vain kääntäjän kannalta välttämättömimmät toiminnot. Lisäksi kirjastossa on ajonaikaista testausta varten luotu Windows Forms -lomake, jonka avulla pystytään monitoroimaan ajonaikaista dataliikennettä sekä saamaan aikaiseksi tapahtumia Jypeli-kirjaston ulkopuolelta. Adapterikirjaston ydin, eli staattinen Adapteri-luokka sisältää seraavat metodit:

- Fysiikkaobjektin Body-olion muodon alustus ja World-olioon liittäminen.
- Fysiikkaobjektien jaottelu kinemaattisiin ja dynaamisiin kategorioihin.
- Kitka-, kimmoisuus-, kulmahidastuvuus-, lineaarihidastuvuuskertoimien välittäminen Body-objektien välillä.
- Fysiikkaobjektin määrittäminen fysiikan laeista piittaamattomaksi, mikäli sitä käytetään sensorina.
- Fysiikkamoottorissa tapahtuvan törmäyksen kytkentä Jypeli-kirjastoon.
- Fysiikkaobjektin alkupaikka- ja alkunopeusvektorit sekä kulma-asema.
- Jypeli-kirjaston Body-luokan sekä Farseer physics Body-luokan toisiinsa kytkentä.
- Tapahtumien välitys fysiikkamoottorista Jypeli-kirjastoon.
- Painovoiman lisäämismetodi.
- Voimavektorien lisäämismetodi.

- Voimaimpulssin lisäämismetodi.
- Fysiikkamoottorin päivittäminen aikaintervallin avulla sekä paikka- ja nopeustietojen päivitys fysiikkaolioiden välillä.

Staattinen versio soveltuu rakenteeltaan erinomaisesti testaamiseen ja uuden fysiikkamoottorin käyttöönottoon. Ei ole itsestäänselvää, että uusi käyttöönotettava fysiikkamoottori mukailee rakenteeltaan edellistä käytettyä. Tässä tutkielmassa käyttöönotettu fysikkamoottori sisälsi poikkeuksellisen paljon yhteneväisyyttä edeltäjäänsä. Staattisen `Adapteri`-luokan päivitysmetodin sisällä tapahtuvat nopeus-, paikka- ja kulmatietojen välitysrutiinit eivät juurikaan aiheuttaneet odotettua pelin hidastumista. Staattisen `Adapteri`-luokan lähdekoodi on esitelty liitteessä C.

4.3 Toteutusvaiheen oliomallinen versio

Kuten jo edellä todettiin, oliomallisen adapterikirjaston lopullisen rakenteen voi hyvinkin pitkälle määritellä liitettävän fysiikkamoottorin rakenne. Esimerkiksi mikäli fysiikkamoottori olisikin muodostettu pelkästään fysiikkaobjektien linkitetystä listasta, jossa listan ensimmäinen objekti olisi ainoa saatavilla oleva rajapinta, voisi Jypeli-kirjaston kytkemisestä tällaiseen fysiikkamoottoriin tulla melko työläs projekti. Oliomallisen adapterikirjaston rakenne kehittyy testaamisen ja käyttöönoton yhteydessä, ja näin ollen on vaikeaa esitellä täydellinen ja optimaalinen luokkarakenne. Tällä hetkellä adapterirajapinta muodostaa seuraavanlaisen käsityksen yleisestä fysiikkamoottorista:

- Fysiikkamaailma, joka sisältää kaiken fysiikan mallintamiseen liittyvän. Näitä ovat esimerkiksi fysiikkaobjektit, törmäystapahtumien kiinniotot, alustustoimet, lisäykset, poistot ja päivitykset. Jypelin aktuaattorina on `PhysicsGame` -luokka ja FarseerPhysicsin sensorina toimii `World`-luokka.
- Fysiikkaobjekti, joka sisältää kaiken yksittäiseen fysiikkaobjektiin liittyvän. Näitä ovat esimerkiksi fysiikkaobjektin muoto, massa, kimmoisuus, paikkavektori, nopeusvektori, vaikuttavien voimien vektorit, kulma-asema ja inertia. Jypelin aktuaattorina on `PhysicsObject`-luokka ja FarseerPhysicsin sensorina on `Body`-luokka.
- Perusmuodot, joita ovat ellipsi, suorakulmio ja monikulmio. Ellipsin erikoistapauksena on ympyrä, mikä on hyvin yleinen ja käyttökelpoinen fysiikkaobjek-

tin muoto. Jypelin ja FarseePhysicsin vastineluokkien määrittely on triviaalia.

Lopuun asti hiottu oliomallinen adapterikirjasto muodostaa ohuen rajapinnan Jypeli-kirjaston ja halutun fysiikkamoottorin välille. Toiminnallisuuden kannalta kaikki ajonaikaiset rutiinit pyritään pitämään mahdollisimman yksinkertaisina, ja lähdekoodissa esiintyvien if-lauseiden määrä mahdollisimman vähäisenä. Alustustoimintojen kannalta muutaman sekunnin viive ei ole merkittävä, mikäli sillä saavutetaan parempaa lähdekoodin ylläpidettävyyttä.

4.4 Jypeli-kirjaston rakenne ennen adapterikirjastoa

Jypeli-kirjastosta ei ole toistaiseksi olemassa arkkitehtuurisuunnitelmaa, mutta Visual Studio 2010 -kehitysympäristössä voi tehdä yksinkertaisia riippuvuusanalyysejä lähdekoodikirjastoihin. Liitteessä A kuvataan Jypeli-kirjaston nimiavaruuksien välisiä riippuvuuksia ennen adapterikirjaston toteuttamista. Riippuvuuksien kuvaaminen siten, että näytettäisiin nimiavaruuksien välisien riippuvuuksien sijaan luokkien välisiä riippuvuuksia, tekisi kaaviosta hankalaselkoisen runsaan luokkien lukumäärän runsauden vuoksi. Pong-pelin peliluokka toteutettuna ilman adapterikirjastoa näyttää seuraavanlaiselta:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using Jypeli;
using Jypeli.Assets;
using Jypeli.Controls;
using Jypeli.Effects;
using Jypeli.Widgets;

public class Peli : PhysicsGame
{
    Vector nopeusYlos = new Vector(0, 800);
    Vector nopeusAlas = new Vector(0, -800);
```

```
    PhysicsObject pallo;
    PhysicsObject maila1;
    PhysicsObject maila2;

    PhysicsObject vasenReuna;
    PhysicsObject oikeaReuna;

    IntMeter pelaajan1Pisteet;
    IntMeter pelaajan2Pisteet;

    public override void Begin()
    {
        LuoKentta();
        AsetaOhjaimet();
        LisaaLaskurit();
        AloitaPeli();
    }

    void LuoKentta()...

    PhysicsObject LuoMaila(double x, double y)...

    void LisaaLaskurit()...

    IntMeter LuoPisteLaskuri(double x, double y)...

    void KasittelePallonTormays(PhysicsObject pallo,
        PhysicsObject kohde)...

    void AloitaPeli()...

    void AsetaOhjaimet()...

    void AsetaNopeus(PhysicsObject maila, Vector nopeus)...
}
```

Peli-luokassa tapahtuu kaikki pelilogiikalle olennainen määrittely. Peli-luokka peritään tässä tapauksessa fysiikkapeli-luokasta. Luokka voitaisiin periä myös kantaluokasta Game, mutta tässä tutkielmassa käsittelemme ainoastaan fysiikkapelejä. Begin-metodi suoritetaan peli-luokasta ensimmäisenä, ja sen täytyy siis sisältää luokan muiden metodien kutsut (tai siinä kutsutuissa metodeissa pitää olla muiden luokassa määriteltyjen metodien kutsut). Peli-luokassa luoduille fysiikkaobjekteille annetaan luonnin yhteydessä ennalta määritellyt alustusarvot, joita voi muokata objektin luomisen jälkeen itse haluamakseen.

4.5 Jypeli-kirjaston rakenne adapterikirjaston jälkeen

Adapterikirjasto toteutettiin Jypeli-kirjaston ja Fysiikkamoottorin väliin liitteessä B olevan nimiavaruuksien riippuvuuskaavion mukaisesti. Molempiin fysiikkamoottoreihin suunnatut adapterikirjastot sisältävät samannimisiä luokkia, ja tästä syystä muutoksia tehdessä täytyy noudattaa suurta varovaisuutta välttääkseen ristiinviittaukset. MathHelper-kirjaston piti alunperin sulautua Physics2DDotNet-kirjastoon, mutta johtuen sen ja Jypeli-kirjaston välisestä korkeasta kytkösasteesta, MathHelper-kirjasto pidetään omana erillisenä kokonaisuutenaan. Pong-pelin peliluokka adapterikirjaston kanssa näyttää seuraavanlaiselta:

```
//Valitse vain yksi alla olevista fysiikkamoottoreista.
//#define PHYSICS2D
#define FARSEER

//Using-riveistä on poistettu kohta "using Jypeli.Assets;".
using System;
using System.Collection.Generic;
using System.Linq;
using System.Text;
using Jypeli;
using Jypeli.Controls;
using Jypeli.Effects;
using Jypeli.Widgets;

#if PHYSICS2D
```

```
using AdapteriPhysics2D;
using AdapteriPhysics2D.Assets;
#elif FARSEER
using AdapteriFarseer;
using AdapteriFarseer.Assets;
#endif

public class Peli : PhysicsGame
{
    Vector nopeusYlos = new Vector(0, 800);
    Vector nopeusAlas = new Vector(0, -800);

    PhysicsObject pallo;
    PhysicsObject maila1;
    PhysicsObject maila2;

    PhysicsObject vasenReuna;
    PhysicsObject oikeaReuna;

    IntMeter pelaajan1Pisteet;
    IntMeter pelaajan2Pisteet;

    public override void Begin()
    {
         //Toistaiseksi pakollinen "silmänkääntötemppu"
        Game.Instance.Level = Level;

        LuoKentta();
        AsetaOhjaimet();
        LisaaLaskurit();
        AloitaPeli();
    }

    void LuoKentta()...
```

```
    PhysicsObject LuoMaila(double x, double y)...

    void LisaaLaskurit()...

    IntMeter LuoPisteLaskuri(double x, double y)...

    void KasittelePallonTormays(PhysicsObject pallo,
        PhysicsObject kohde)...

    void AloitaPeli()...

    void AsetaOhjaimet()...

    void AsetaNopeus(PhysicsObject maila, Vector nopeus)...
}
```

Adapterikirjaston käyttö edellyttää peliluokalta ainoastaan muutaman rivin lisäämistä, ja yhden poistoa. Erona alkuperäiseen Jypeli-kirjastoon `Assets`-nimiavaruus on siirretty adapterikirjaston alle. Peliluokka peritään `PhysicsGame`-luokasta, joka on adapterikirjastossa määritelty erikseen käytettävän fysiikkamoottorin kanssa yhteensopivaksi. `Level`-luokka on katkaistu siten, että kussakin adapterikirjastossa oleva `PhysicsLevel`-luokka peritään yhteisestä Jypeli-kirjastossa olevasta `Level`-kantaluokasta. Tästä johtuen täytyy `Begin`-metodin alussa määritellä globaali muuttuja `Game.Instance.Level` `PhysicsGame`-luokan attribuutin `Level`-attribuutiksi. Käytettävän fysiikkamoottorin määrittely tapahtuu yksinkertaisesti define-määrittelyjen avulla siten, että käytetään halutun adapterin nimiavaruuksia.

4.6 Adapterikirjaston keskeisimmät luokat

Adapterikirjasto sisältää useita kymmeniä luokkia, joista suurimman osan tehtävänä on pitää luokkien perintähierarkia siistinä. Adapterikirjaston toiminnallisuuden kannalta oleellisimmat luokat ovat `PhysicsGameBase` (liitteet D ja G), `PhysicsGame` (liitteet E ja H) ja `PhysicsObject` (liitteet F ja I). Lähdekoodin rivien määrän suuruuden vuoksi luokat esitellään siten, että vain adapterikirjaston kannalta relevantit metodit ja alustajat ovat kirjoitettuna auki. Kaikista muista lähdekoodin osista esitellään vain määrittely- ja kommenttirivit.

5 Johtopäätökset

Tutkielman teoriaosassa pyrittiin avartamaan fysiikkamoottorin ymmärtämystä. Konstruktiivisessa osassa osoitettiin fysiikkamoottorin vaihdon olevan perusteltua, esitettiin joitain argumentteja uuteen sovitettavaan fysiikkamoottoriin liittyen sekä perustellusti toteutettiin adapterikirjasto Jypeli-kirjaston ja fysiikkamoottorin välille. Adapterikirjastoa ei toteutettu loppuun asti, sillä lopullinen käyttöönotto edellyttää perusteellisempaa testausta, kuin mitä tämän tutkielman puitteissa on suoritettu. Sen sijaan adapterikirjasto pyrkii tarjoamaan vaivattoman siirtymisen fysiikkamoottorien välillä, jotta kirjaston koeluontoinen käyttöönotto olisi mielekkäämpää.

5.1 Tutkielman tavoitteiden toteutuminen

Tämän tutkielman eteneminen perustui asetettujen välitavoitteiden saavuttamisen kautta uusiin tavoitteisiin siirtymällä. Tällainen etenemisperiaate oli perusteltua siksi, että ensin oli tutkittava se, että onko olemassa sopivaa fysiikkamoottoriehdokasta, ja että onko ylipäänsä mahdollista edes ottaa uutta fysiikkamoottoria käyttöön. Mikäli alkuvaiheissa olisi todettu pelkän uuden fysiikkamoottorin testaamisen johtavan koko Jypeli-kirjaston uudelleenkirjoittamiseen, olisi tutkielman rakenne ollut täysin toisenlainen. Tässä luvussa esitellään ne kolme välitavoitetta, joiden saavuttaminen vaikutti oleellisesti koko tutkielman etenemiseen.

5.1.1 Uuden fysiikkamoottorin valinta

Uuden fysiikkamoottorin valintaan vaikuttivat suurimmalta osin lisenssiehdot sekä kyseisellä fysiikkamoottorilla toteutetut malliesimerkit. Myös keskustelupalstojen suosittelulla ja mielipiteillä oli painoarvoa valintaa tehdessä. Fysiikkamoottorin sisäisellä luokkarakenteella ei olisi ollut väliä, mutta ensimmäiseksi sovitettavaksi fysiikkamoottoriksi valittu Farseer Physics sisälsi odotettua yhtenäisemmän rakenteen, kuin edeltäjänsä. Ensisijaisesti sovitettavaa fysiikkamoottoria etsittiin 2D-peleihin erikoistuneiden kirjastojen joukosta, mutta harkinnan varassa olisi voinut olla myös 3D-peleihin suunnattu fysiikkamoottori. Myös itse toteutetun arcade-tyylisen fysiikkamoottorin käyttöönotto olisi voinut tulla kysymykseen.

5.1.2 Uuden fysiikkamoottorin koekäyttöönotto

Fysiikkamoottorin vaihdon tarve syntyi niistä fysiikan mallinnuksen virheistä, joita aikaisemmassa fysiikkamoottorissa ilmeni. Tällaisia virheitä olivat muun muassa epämääräiset kimpoamiskulmat sekä kitkattomien fysiikkaobjektien toisiinsa tarttuminen. Ensimmäisten keskeneräisen adapterikirjaston kautta tehtyjen testitapausten perusteella voitiin todeta aikaisemmin esiintyneiden virheiden poistuminen. Uuden fysiikkamoottorin käyttöönoton yhteydessä oli havaittavissa jotain eroavaisuuksia fysiikan mallinnuksessa. Esimerkiksi samalla voimalla samanpainoiset fysiikkaobjekti lensivät erilaisen lentoradan. Kyseessä voi olla jokin skaalaus- tai ilmanvastuskerroin, jota ei ole otettu huomioon oikealla tavalla.

5.1.3 Adapterikirjaston kehittäminen

Uuden fysiikkamoottorin ja onnistuneiden ensitestien jälkeen oli luontevaa perustella adapterikirjaston kehittely. Adapterikirjaston ensisijainen tehtävä oli mahdollistaa fysiikkamoottorin mielekäs vaihto sen tarjoaman rajapinnan avulla siten, että fysiikan mallinnuksen laatu ei kärsisi. Toissijaisena tehtävänä oli tarjota helppo siirtyminen fysiikkamoottoreiden välillä, mikäli pelikehityksessä todetaan, että jokin asia olisi viisainta toteuttaa eri fysiikkamoottorilla, kuin mitä sillä hetkellä on käytettävissä. Käytännössä tämä tarkoittaa fysiikkamoottorin vaihtamista lähdekoodissa vain yhtä riviä muuttamalla. Uuden fysiikkamoottorin liittäminen adapteriin on työlästä riippuen liitettävän fysiikkamoottorin rakenteen ja toiminnan kompleksisuudesta.

5.2 Adapterikirjaston jatkokehityksen vaihtoehdot

Tässä tutkielmassa adapterikirjasto kehitettiin siihen pisteeseen, että voitiin todeta fysiikkamoottorin vaihdon vähentävän oleellisesti ei-toivottua virheellistä fysiikan mallinnusta. Adapterikirjaston pidemmälle kehittelyssä täytyy ottaa huomioon myos muiden kehittäjien tarpeet, sekä määritellä adapterikirjaston ensisijainen tehtävä. Vaihtoehtoisia ensisijaisia tehtäviä voisivat olla nopea uuden fysiikkamoottorin käyttöönotto, ensiluokkainen fysiikanmallinnus uudella fysiikkamoottorilla tai adapterikirjaston ja Jypeli-kirjaston korkeatasoinen ylläpidettävyys. Tässä kappaleessa käydään läpi kolme erilaista vaihtoehtoa siihen liittyen, miten Jypeli-kirjaston kehitys voisi edetä adapterikirjaston kehittämisen myötä.

5.2.1 Erilliset Jypeli-kirjastot

Jo alkupään kokeiluprojektien aikana tuli ilmi Jypeli-kirjaston ja fysiikkamoottorin suuri kytkösaste. Adapterikirjaston toteuttaminen sille tasolle, että molemmat fysiikkamoottorit toimisivat siinä täydellisesti, tulisi vaatimaan muutoksia erittäin moneen Jypeli-kirjaston luokkaan. Jokaiselle fysiikkamoottorille suunnattu oma erillinen Jypeli-kirjasto antaisi mahdollisuuden fysiikkamoottorin ominaisuuksien laajaan käyttöönottoon, mutta heikentäisi huomattavasti Jypeli-kirjaston ylläpidettävyyttä. Jokainen muutos Jypeli-kirjastossa vaatisi jokaisen omalle fysiikkamoottorilleen spesifioidun kirjaston päivittämistä erikseen. Tässä vaihtoehdossa uuden fysiikkamoottorin lisääminen muiden rinnalle vaikeuttaisi tilannetta entisestään.

5.2.2 Kehitysprojekteissa toteutetun adapterikirjaston jatkokehitys

Kehitysprojekteissa adapterikirjasto toteutettiin siten, että peliluokassa yhtä riviä vaihtamalla voitiin valita kahden fysiikkamoottorin välillä. Kuitenkaan ihan kaikkia Farseer-fysiikkamoottorin toimintoja ei otettu toistaiseksi käyttöön. Jatkokehityksessä Physics2DDotNet-fysiikkamoottorille toteutetut ominaisuudet vietäisiin yksitellen uuteen fysiikkamoottoriympäristöön. Tässäkään ratkaisussa ei ylimääräiseltä työltä vältytä, kun tehdään uudistuksia Jypeli-kirjastoon. Muutokset, jotka koskevat fysiikkakirjastoa käyttäviä luokkia, joudutaan tekemään erikseen jokaista fysiikkamoottoria varten. Ylläpidettävyys on siis hivenen parempi, kuin mitä erillisen Jypeli-kirjaston toteutusmallissa, mutta tämäkään vaihtoehto ei tue rinnakkaisten fysiikkamoottorien määrän kasvattamista.

5.2.3 Jypeli-kirjaston uudelleenkirjoitus

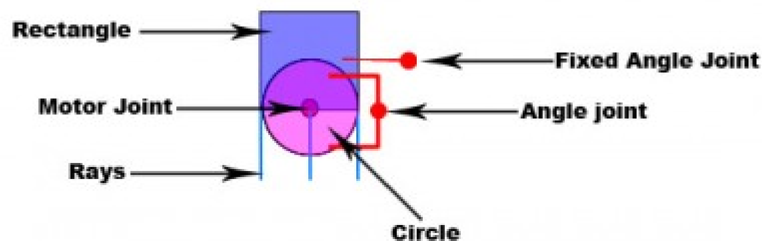
Aikoinaan, kun Jypeli-kirjaston kehitys aloitettiin, ei sille oltu asetettu kovin tarkkoja vaatimuksia. Vaatimukset kasvoivat kehityksen myötä, ja uutta rakennettiin aina vanhan päälle. Myös kehitysympäristö ja ohjelmointikieli valittiin sen aikaisten tarpeiden mukaisesti. Tänä päivänä ja tulevaisuudessa kehitystyökalujen yhteensopivuus käytettyjen teknologioiden kanssa ei ole enää varmaa. Tässä vaiheessa voisi olla perusteltua siirtyä kokonaan itsenäiseen Jypeli-kirjaston kehitykseen käyttäen ainoastaan avoimen lähdekoodin sovelluksia ja työkaluja. Tähän asti saavutetun kokoemuksen ja tiedon turvin pystyttäisiin määrittelemään melko kattavasti uuden Jypeli-kirjaston vaatimukset sekä arkkitehtuuri.

6 Projektin aikana esiin tulleet jatkokehitysideat

Tiedon haun, ohjausten ja itsenäisen tutkiskelun myötä tuli tutkielman aikana useita varsinaisesta aiheesta eroavia kehitysideoita. Jotkin ideoista voisivat jopa olla varteenotettavia erillisen tutkielman aiheita. Ideat ovat rajattu siten, että ne eivät varsinaisesti liity tutkielman aiheeseen, mutta niissä täytyy olla jokin piirre, joka jollain lailla kytkeytyy nuorten peliohjelmoinnin piiriin.

6.1 Boxboy-tyylinen tasohyppelyhahmo

Fysiikan mallinnuksen ongelmia määriteltäessä käytettiin yhtenä tapauksena tasohyppelypelin hahmoa. Boxboy-tyylisessä tasohyppelypelihahmossa fyysinen rakenne koostuu kahdesta osasta, varsinaisesta suorakaiteen muotoisesta rungosta ja ympyränmuotoisesta vetopyörästä. Boxman-hahmon idea on siinä, että sen avulla voidaan vähentää dynaamisen fysiikkaobjektin siirtelyä pelkkien nopeusvektorien avulla. Boxboy-hahmo saa liike-energiansa vetopyörästä, ja sen kehon asento pidetään pystysuorassa esimerkiksi äärettömän suuren inertian avulla. Vetopyörän on oltava hahmoa kapeampi, ettei tapahdu odottamattomia ilmiöitä, kuten esimerkiksi pystysuoraa seinää pitkin kiipeämistä.



Kuva 6.1: Boxboy-tyylisen tasohyppelyhahmon periaate [5].

6.2 C-kielisiä fysiikkamoottoreita tukeva adapterikirjasto

Käyttöönotettavien fysiikkamoottoreiden rajaus tämän tutkielman aikana oli melkoisen tiukka. Sen piti tukea XNA-pelikehystä, sen piti olla C#-kielellä kirjoitettu, ja sen piti olla avoimen lähdekoodin lisenssin alainen. Kuitenkin parhaat pelimoottorit tehdään niin lähellä konekielistä rajapintaa kuin vain mahdollista. Jos fysiikkamoottorin rajauksesta voitaisiin pudottaa pois C#-kielisyys ja XNA-yhteensopivuus, saataisiin otannaksi paljon suurempi fysiikkamoottoreiden valikoima. Tällaisen ominaisuuden toteuttaminen tulisi ajankohtaiseksi, mikäli päätettäisiin kirjoittaa Jypeli-kirjasto kokonaan uudestaan. Siinä tapauksessa adapterikirjaston tyyli tulisi muuttumaan huomattavasti yleistävämmäksi ja universaalimmaksi [1].

6.3 Rinnakkaisohjelmoinnin tuominen Jypeli-kirjastoon

Ohjelmointitekniikassa yleisesti muotiin tullut rinnakkaisohjelmointi voisi tuoda uusia mahdollisuuksia myös Jypeli-maailmaan. Erityisesti fysiikan mallinnus voisi olla soveltuva rinnakkaisuuden kohde. Esimerkiksi voitaisiin käsitellä x- ja y -akselien suuntaiset fysiikan tapahtumat omissa säikeissään. Kolmiulotteisissa pelissä voitaisiin vastaavasti toimia kolmella säikeellä. Myös törmäystarkastelu voitaisiin toteuttaa omassa säikeessään. Rinnakkaisohjelmointi on kuitenkin erittäin haastavaa, eikä tehokkaita esimerkkitoteutuksia tunnu olevan saatavilla. Kuitenkin rinnakkaisohjelmointi antaisi ihan uusia mahdollisuuksia esimerkiksi näytönohjaimen prosessoriresurssien hyödyntämiseen.

6.4 Raycasting-tekniikkaa käyttävien 3D-pelien kehittely

Aikanaan tutuksi tulleet Doom ja Duke Nukem 3D -pelit toivat raycasting-tekniikan päivänvaloon. Sen aikaisissa tietokoneissa kyseinen tekniikka oli välttämätöntä sujuvan pelattavuuden aikaansaamiseksi. Tämän päivän tietokoneet kykenevät käsittelemään kolmiulotteisia objekteja niin vaivattomasti, ettei raycasting-tekniikalle ole enään todellista tarvetta. Sen sijaan yhdistämällä helppokäyttöinen kaksiulotteisen tasohyppelypelin kenttäeditori raycasting-moottoriin, voitaisiin ainakin teaoriassa luoda helposti ja nopeasti kolmiulotteisia fysiikkapelejä. Ehkä kevyemmällä 3D-grafiikan käsittelyllä saavutetaan parempaa pelattavuutta mobiililaitteissa, joissa suoritintehot eivät ainakaan toistaiseksi vastaa pc-tietokoneiden tasoa. [18]

6.5 Useamman pistemäisen painovoimakentän maailma

Tällä hetkellä fysiikkamoottorit käyttävät pääsääntöisesti painovoimavektoria kuvaamaan painovoimaa. Suurimmassa osassa pelejä se riittää tuomaan tarvittavan todentuntuisuuden, varsinkin jos pelimaailma liittyy johonkin maan päällä tapahtuvaan. Kun siirrytään ilmakehästä avaruuteen, tulee tarkastelun piiriin useammat pistemäiset painovoimakeskukset. Nämä keskukset sijaitsevat eri taivaankappaleiden keskikohdissa. Nykyisillä fysiikkamoottoreilla voidaan toki saada aikaiseksi vastaava ilmiö, mutta sen toteuttaminen edellyttää enemmän tai vähemmän keinotekoisia ratkaisuja. Pistemäisellä painovoimakeskuksella voitaisiin toteuttaa myös perinteisiä maan päälle sijoittuvia tasohyppelypelejä asettamalla painovoimapiste tuhansien kilometrien syvyyteen maan pinnasta.

7 Yhteenveto

Fysiikkamoottorit pelikehityksessä ovat ehkä ohjelmistotekniikan yksiä haastavimpia tutkimusalueita. Fysiikkamoottorien teoriaa voidaan kuvata matemaattisesti, ja matematiikan käsittein voidaan luoda monenlaisia ideaalisia fysiikkamoottorien periaatemalleja. Kuitenkin teorian vieminen käytäntöön konemaailmassa ei ole niin suoraviivaista. Esimerkiksi yhtäaikaisuuden aikaansaaminen nykypäiväisen tietokoneen prosessoriarkkitehtuurissa ei ole lainkaan triviaalia. Ennen pitkää tullaan siihen pisteeseen, että teorian käytäntöön soveltamisen sijaan yritetään kehitellä algoritmeja erilaisten virheiden ja epätarkkuuksien korjaamiseen. Tässä tutkielmassa siis tarkasteltiin hieman fysiikan mallintamisen teoriaa, ja osoitettiin se, että pelikehityksessä voidaan fysiikkamoottorin vaihtamisella saada aikaan merkittäviä parannuksia fysiikan mallintamiseen. Mitään ideaalisen fysiikkamoottorin rakennetta tai arkkitehtuuria ei esitetty. Fysiikkamoottorin vaihtamisen vaikutusten johdosta voitiin perustella tämän tutkielman konstruktiivisessa osassa toteutettu adapterikirjasto.

Adapterikirjasto toteutettiin niissä aikataulullisissa ja ohjelmointitaidollisissa rajoissa, mitä tekijällä oli sillä hetkellä käytössään. Lähtötilanteena Jypeli-kirjaston ymmärtämys rajoittui siihen rajapintaan, mitä käytetään erilaisten pelien kehittämisessä. Lopputuloksena ainakin Pong-peli pystytään toteuttamaan varmasti uuden fysiikkamoottorin turvin, mutta vähänkään monimutkaisemmissa toteutuksissa voi tulla vielä yllättäviä virhetilanteita vastaan. Tutkimuksen ohessa tapahtui myös oppimista, jonka turvin tutkimusta voitiin syventää edelleen. Adapterikirjasto tullaan ottamaan käyttöön nuorten peliohjelmointikurssilla, ja mahdollinen jatkokehitys tullaan tekemään käyttäen lähtökohtana tämän tutkielman tuloksia.

8 Lähteet

[1] Adrian Boeing. *Physics Abstraction Layer*. http://www.adrianboeing.com/pal/index.html. Viitattu 27.6.2012.

[2] David M. Bourg. *Physics for Game Developers*. O'Reilly Media, 1005 Gravenstein Highway North, Sebastopol, CA 95472, 2002.

[3] Erin Catto. *Box2D, A Physics Engine for Games*. http://box2d.org/. Viitattu 27.2.2012.

[4] Nimimerkki DeanoC ja nimimerkki noone88. *JigLibX Physic Library*. http://jiglibx.codeplex.com/. Viitattu 26.2.2012.

[5] Bryan Dismas. *Platformer Character Control (Farseer Physics Engine)*. http://amazingretardo.simiansoftwerks.com/2010/02/17/platformer-character-control-farseer-physics-engine/. Viitattu 27.6.2012.

[6] Erich Gamma, Richard Heim, Ralph Johnson, ja John Vlissides. *Olio-ohjelmointi Suunnittelumallit*. Addison-Wesley, United States, 2000. Suomennos 2001, IT Press.

[7] Kurt Jaegers. *XNA 4.0 Game Development by Example*. Packt Publishing Ltd., 32 Lincoln Road, Olton, Birmingham, B27 6PA, UK., 2010.

[8] Kai Koskimies ja Tommi Mikkonen. *Ohjelmistoarkkitehtuurit*. Talentum Media Oy, Helsinki, 2005.

[9] Antti-Jussi Lakanen. *Nuorten peliohjelmointi*. Jyväskylän yliopisto, tietotekniikan laitos, Jyväskylä, 2010.

[10] André LaMothe, John Ratcliff, Mark Seminatore, ja Denise Tyler. *Peliohjelmointi - Tehokäyttäjän opas*. Sams Publishing, 800 East 96th Street, Indianapolis, Indiana 46240, 1994. Suomennos 1997, Suomen Atk-kustannus Oy.

[11] Tom Miller ja Dean Johnson. *XNA Game Studio 4.0 Programming*. Addison-Wesley, United States, 2011.

[12] Ian Millington. *Game Physics Engine Development*. Elsevier Inc., 30 Corporate Drive, Suite 400, Burlington, MA 01803, USA, 2010.

[13] Ian Millington, Christer Ericson, David Eberly, Russell Smith, ja Danny Chapman. *Oops! 3D Physics Framework (and more) for XNA*. http://oopsframework.codeplex.com/. Viitattu 26.2.2012.

[14] Grant Palmer. *Physics for Game Programmers*. Apress, 2560 Ninth Street, Suite 219, Berkeley, CA 94710, 2005.

[15] Jono Porter. *Physics2D.Net: A 2D Physics Engine Written in C#*. http://code.google.com/p/physics2d/. Viitattu 25.2.2012.

[16] Rudy Rucker. *Software Engineering and Computer Games*. Addison-Wesley, United States, 2003.

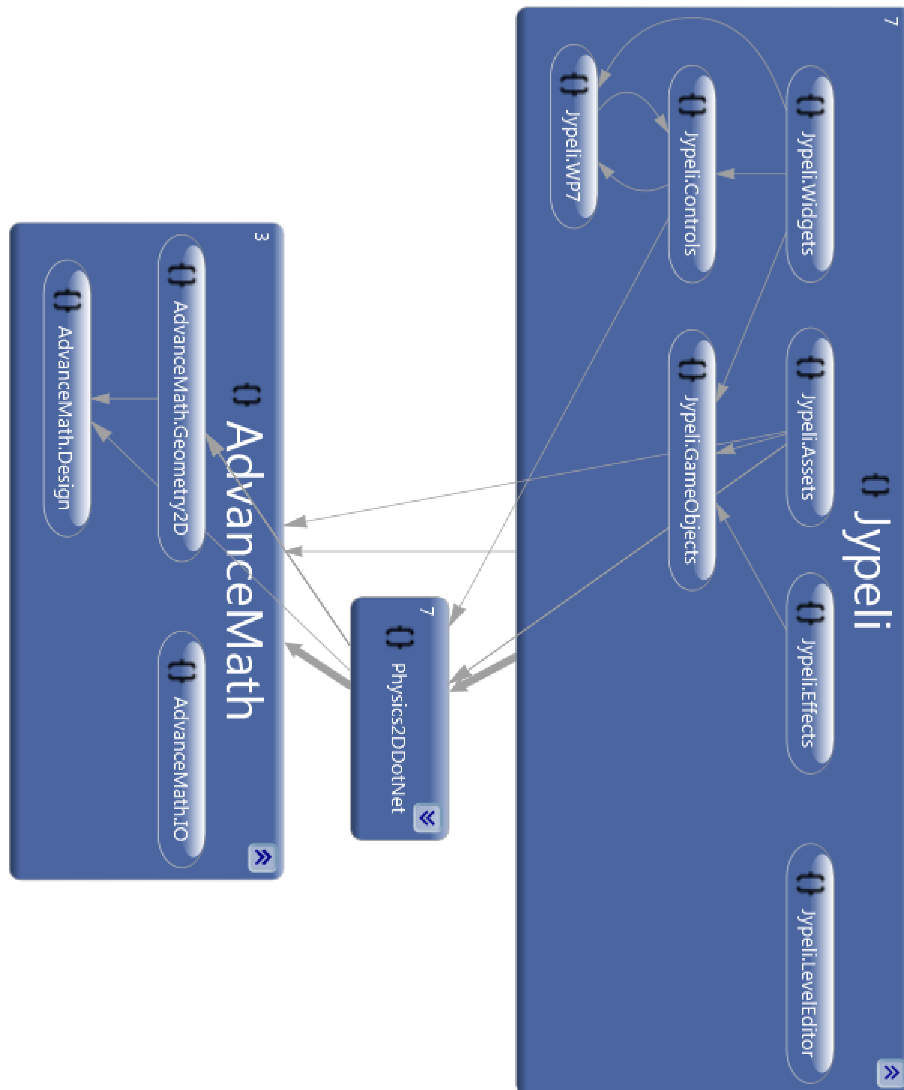
[17] Wendy Stahler. *Beginning Math and Physics for Game Programmers*. New Riders, 800 East 96th Street, 3rd Floor, Indianapolis, Indiana 46240, 2004.

[18] Lode Vandevenne. *Raycasting*. http://lodev.org/cgtutor/raycasting.html. Viitattu 27.6.2012.

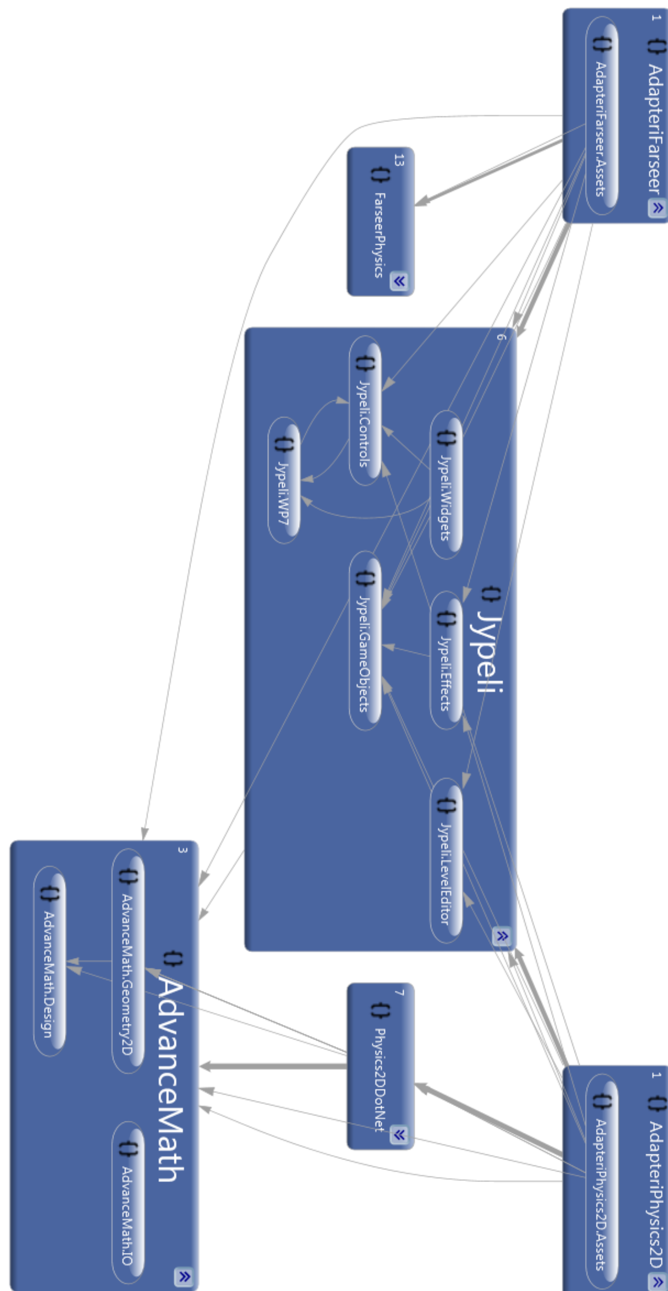
[19] Nimimerkki vankaandreev. *3D Graphics for Windows Phone 7 Using the XNA Framework*. 2.4.2011. http://www.codeproject.com/Articles/176162/3D-Graphics-for-Windows-Phone-7-Using-the-XNA-Fram. Viitattu 25.2.2012.

[20] Jeff Weber ja Ian Qvist. *Farseer Physics*. http://farseerphysics.codeplex.com/. Viitattu 27.2.2012.

A Jypeli-kirjasto ennen adapterikirjastoa



B Jypeli-kirjasto adapterikirjaston jälkeen



C Staattinen Adapteri-luokka

```
using Microsoft.Xna.Framework;
using System.Collections.Generic;

namespace Adapterikirjasto
{
    /// <summary>
    /// Staattinen Adapteri-luokka vastaa kaikesta Jypeli-kirjaston ja fysiikkamoottorin
    /// välisestä dataliikenteestä.
    /// </summary>
    public static class Adapteri
    {
        private static FarseerPhysics.Dynamics.World farseerWorld = new FarseerPhysics.Dynamics.World ↙
    (Vector2.Zero);
        //private static Monitori monitori = new Monitori(farseerWorld);

        /// <summary>
        /// Body-objektin lisääminen fysiikkamaailmaan.
        /// </summary>
        /// <param name="adapterBody"></param>
        public static void lisaaBody(Body adapterBody)
        {
            FarseerPhysics.Dynamics.Body farseerBody;
            if (adapterBody.Shape.GetType() == typeof(CircleShape))
            {
                CircleShape circleShape = (CircleShape)adapterBody.Shape;
                farseerBody = FarseerPhysics.Factories.BodyFactory.CreateCircle(farseerWorld, (float) ↙
    circleShape.Radius, 1f);
            }
            else
            {
                Vector2D[] vertexes = adapterBody.Shape.Vertexes;
                Vector2[] lista = new Vector2[vertexes.Length];
                for (int i = 0; i < vertexes.Length; i++)
                {
                    lista[i] = new Vector2((float)vertexes[i].X, (float)vertexes[i].Y);
                }
                FarseerPhysics.Common.Vertices vertices = new FarseerPhysics.Common.Vertices(lista);
                farseerBody = FarseerPhysics.Factories.BodyFactory.CreatePolygon(farseerWorld, vertices, 1f) ↙
    ;
            }

            if (double.IsPositiveInfinity(adapterBody.Mass.Mass))
            {
                farseerBody.BodyType = FarseerPhysics.Dynamics.BodyType.Kinematic;
                farseerBody.Mass = float.PositiveInfinity;
            }
            else if (adapterBody.Mass.Mass > 0)
            {
                farseerBody.BodyType = FarseerPhysics.Dynamics.BodyType.Dynamic;
                farseerBody.Mass = (float)adapterBody.Mass.Mass;
            }
            else
            {
                farseerBody.BodyType = FarseerPhysics.Dynamics.BodyType.Static;
            }

            if (double.IsPositiveInfinity(adapterBody.Mass.MomentOfInertia)) farseerBody.Inertia = float. ↙
    PositiveInfinity;

            farseerBody.Friction = (float)adapterBody.Coefficients.DynamicFriction;
            farseerBody.Restitution = (float)adapterBody.Coefficients.Restitution;
            farseerBody.AngularDamping = (float)adapterBody.AngularDamping;
            farseerBody.LinearDamping = (float)adapterBody.LinearDamping;
            farseerBody.IgnoreGravity = adapterBody.IgnoresGravity;
            farseerBody.IsSensor = adapterBody.IgnoresCollisionResponse;
            farseerBody.OnCollision += tapahtuma;
            farseerBody.LinearVelocity = new Vector2((float)adapterBody.State.Velocity.X, (float)adapterBody ↙
    .State.Velocity.Y);
            farseerBody.Position = new Vector2((float)adapterBody.State.Position.X, (float)adapterBody.State ↙
    .Position.Y);
            farseerBody.Rotation = (float)adapterBody.State.Position.Angular;
            farseerBody.UserData = adapterBody;
            adapterBody.adapteeBody = farseerBody;
        }
```

```
        /// <summary>
        /// Törmäystapahtuman välittäminen Body-objektille.
        /// </summary>
        /// <param name="collider"></param>
        /// <param name="collidee"></param>
        /// <param name="contact"></param>
        /// <returns></returns>
        public static bool tapahtuma(FarseerPhysics.Dynamics.Fixture collider, FarseerPhysics.Dynamics.  ↙
    Fixture collidee, FarseerPhysics.Dynamics.Contacts.Contact contact)
        {
            Body adapterBodyCollider = (Body)collider.Body.UserData;
            Body adapterBodyCollidee = (Body)collidee.Body.UserData;
            adapterBodyCollider.OnCollision(null, adapterBodyCollidee, null);
            return true;
        }

        /// <summary>
        /// Voiman lisääminen.
        /// </summary>
        /// <param name="voima"></param>
        /// <param name="adapterBody"></param>
        public static void lisaaVoima(Vector2D voima, Body adapterBody)
        {
            if (adapterBody.adapteeBody != null)
            {
                FarseerPhysics.Dynamics.Body farseerBody = (FarseerPhysics.Dynamics.Body)adapterBody.  ↙
    adapteeBody;
                farseerBody.ApplyForce(new Vector2((float)voima.X, (float)voima.Y));
            }
        }

        /// <summary>
        /// Impulssin lisääminen.
        /// </summary>
        /// <param name="impulssi"></param>
        /// <param name="adapterBody"></param>
        public static void lisaaImpulssi(Vector2D impulssi, Body adapterBody)
        {
            if (adapterBody.adapteeBody != null)
            {
                FarseerPhysics.Dynamics.Body farseerBody = (FarseerPhysics.Dynamics.Body)adapterBody.  ↙
    adapteeBody;
                farseerBody.ApplyLinearImpulse(new Vector2((float)impulssi.X, (float)impulssi.Y));
            }
        }

        /// <summary>
        /// Painovoiman lisääminen.
        /// </summary>
        /// <param name="painovoima"></param>
        public static void lisaaPainovoima(Vector2D painovoima)
        {
            farseerWorld.Gravity = new Vector2((float)painovoima.X, (float)painovoima.Y);
        }

        /// <summary>
        /// Päivitysmetodi, jota kutsutaan 60 kertaa sekunnissa.
        /// </summary>
        /// <param name="intervalli"></param>
        public static void paivita(double intervalli)
        {
            Body adapterBody;
            List<FarseerPhysics.Dynamics.Body> farseerBodies = farseerWorld.BodyList;

            foreach (FarseerPhysics.Dynamics.Body farseerBody in farseerBodies)
            {
                adapterBody = (Body)farseerBody.UserData;
                farseerBody.Position = new Vector2((float)adapterBody.State.Position.X, (float)adapterBody.  ↙
    State.Position.Y);
                farseerBody.Rotation = (float)adapterBody.State.Position.Angular;
            }

            farseerWorld.Step((float)intervalli);
```

```
            foreach (FarseerPhysics.Dynamics.Body farseerBody in farseerBodies)
            {
                adapterBody = (Body)farseerBody.UserData;
                if (farseerBody.BodyType == FarseerPhysics.Dynamics.BodyType.Kinematic)
                    farseerBody.LinearVelocity = new Vector2((float)adapterBody.State.Velocity.X, (float)    ↙
    adapterBody.State.Velocity.Y);
                if (farseerBody.BodyType == FarseerPhysics.Dynamics.BodyType.Dynamic)
                    farseerBody.LinearVelocity = new Vector2((float)adapterBody.State.Velocity.X,            ↙
    farseerBody.LinearVelocity.Y);
                adapterBody.State.Position.X = farseerBody.Position.X;
                adapterBody.State.Position.Y = farseerBody.Position.Y;
                adapterBody.State.Position.Angular = farseerBody.Rotation;
            }
        }
    }
}
```

D Farseer Physics -adapterikirjaston PhysicsGameBase-luokka

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using Microsoft.Xna.Framework;
using AdapteriFarseer.Assets;
using Jypeli;

namespace AdapteriFarseer
{
    public abstract class PhysicsGameBase : Jypeli.Game
    {
        protected struct CollisionRecord //ei relevantti

        public static FarseerPhysics.Dynamics.World farseerWorld;
        private SynchronousList<Joint> Joints = new SynchronousList<Joint>();
        protected Dictionary<CollisionRecord, CollisionHandler<IPhysicsObject, IPhysicsObject>>
            collisionHandlers = new Dictionary<CollisionRecord, CollisionHandler<IPhysicsObject,
                IPhysicsObject>>();

        /// <summary>
        /// Onko fysiikan laskenta käytössä vai ei.
        /// </summary>
        public bool PhysicsEnabled { get; set; }

        /// <summary>
        /// Alustaa uuden fysiikkapelin.
        /// </summary>
        /// <param name="device">Mikä monitori käytössä, 1=ensimmäinen</param>
        public PhysicsGameBase(int device)
            : base( device )
        {
            PhysicsEnabled = true;
            farseerWorld = new FarseerPhysics.Dynamics.World(Vector2.Zero);

            Joints.ItemAdded += OnJointAdded;
            Joints.ItemRemoved += OnJointRemoved;
        }

        /// <summary>
        /// Luodaan Farseer Physicsin vastine aikaisemmin käytetylle
        /// nivelliitokselle.
        /// </summary>
        /// <param name="j"></param>
        void OnJointAdded(Joint j)
        {
            j.Lifetime.IsExpired = false;
            if (j.GetType() == typeof(HingeJoint))
            {
                HingeJoint hj = (HingeJoint)j;
                j.FarseerRevoluteJoint = new FarseerPhysics.Dynamics.Joints.RevoluteJoint(
                    hj.Body1, hj.Body2, new Vector2(
                        hj.Body2.Position.X - hj.Body1.Position.X,
                        hj.Body2.Position.Y - hj.Body1.Position.Y),
                        new Vector2(0.0f, 0.0f));
                farseerWorld.AddJoint(j.FarseerRevoluteJoint);
            }
        }

        void OnJointRemoved(Joint j)
        {
            j.Lifetime.IsExpired = true;
            farseerWorld.RemoveJoint(j.FarseerRevoluteJoint);
        }

        protected override void OnObjectAdded(IGameObject obj)
        {
            if (obj is PhysicsObject)
            {
                AddToEngine((PhysicsObject)obj);
            }

            base.OnObjectAdded(obj);
        }
```

```
        protected override void OnObjectRemoved(IGameObject obj)
        {
            if (obj is PhysicsObject)
            {
                RemoveFromEngine((PhysicsObject)obj);
            }

            base.OnObjectRemoved(obj);
        }

        /// <summary>
        /// Pysäyttää kaiken liikkeen.
        /// </summary>
        public void StopAll()
        {
            foreach (var layer in Layers)
            {
                foreach (var obj in layer.Objects)
                {
                    if (obj is PhysicsObject)
                        ((PhysicsObject)obj).Stop();
                }
            }
        }

        /// <summary>
        /// Nollaa kaiken (kontrollit, näyttöobjektit, ajastimet ja fysiikkamoottorin).
        /// </summary>
        public override void ClearAll()
        {
            ClearPhysics();
            base.ClearAll();
        }

        /// <summary>
        /// Nollaa fysiikkamoottorin.
        /// </summary>
        private void ClearPhysics()
        {
            farseerWorld.Clear();
        }

        private void AddToEngine(PhysicsObject po)
        {
            if (po is Projectile) po.Body.IsBullet = true;
        }

        private void RemoveFromEngine(PhysicsObject po)
        {
            farseerWorld.RemoveBody(po.Body);
        }

        /// <summary>
        /// Lisää liitoksen peliin.
        /// </summary>
        public void Add(Joint j)
        {
            Joints.Add(j);
        }

        /// <summary>
        /// Poistaa liitoksen pelistä.
        /// </summary>
        /// <param name="j"></param>
        internal void Remove(Joint j)
        {
            Joints.Remove(j);
        }

        /// <summary>
        /// Poistaa liitoksen pelistä.
        /// </summary>
        /// <param name="j"></param>
```

```
        internal void Remove(AxleJoint j)
        {
            Joints.Remove(j.innerJoint);
        }

        /// <summary>
        /// Lisää liitoksen peliin.
        /// </summary>
        public void Add(AxleJoint j)
        {
            if (!j.Object1.IsAddedToGame) Add(j.Object1);
            if (j.Object2 != null && !j.Object2.IsAddedToGame) Add(j.Object2);

            Add(j.innerJoint);
        }

        public override void Add(IGameObject o, int layer)
        {
            if (o is PhysicsStructure)
            {
                foreach (var joint in ((PhysicsStructure)o).Joints)
                {
                    Add(joint);
                }
            }

            base.Add(o, layer);
        }

        /// <summary>
        /// Ajetaan kun pelin tilannetta päivitetään. Päivittämisen voi toteuttaa perityssä luokassa
        /// toteuttamalla tämän metodin. Perityn luokan metodissa tulee kutsua kantaluokan metodia.
        /// </summary>
        /// <param name="time"></param>
        protected override void Update(Time time)
        {
            double dt = time.SinceLastUpdate.TotalSeconds;

            if (PhysicsEnabled)
            {
                farseerWorld.Step((float)dt);
            }

            base.Update(time);

            // Updating joints must be after base.Update so that the bodies
            // are added to the engine before the joints

            Joints.Update(time);
        }

        /// <summary>
        /// Määrää, mihin aliohjelmaan siirrytään kun olio <code>obj</code> törmää johonkin toiseen olioon.
        /// </summary>
        /// <typeparam name="T">Kohdeolion tyyppi.</typeparam>
        /// <param name="obj">Törmäävä olio</param>
        /// <param name="handler">Törmäyksen käsittelevä aliohjelma.</param>
        public void AddCollisionHandler<O, T>(O obj, CollisionHandler<O, T> handler)
            where O : IPhysicsObject
            where T : IPhysicsObject //ei relevantti

        /// <summary>
        /// Määrää, mihin aliohjelmaan siirrytään kun yleinen fysiikkaolio <code>obj</code>
        /// törmää johonkin toiseen yleiseen fysiikkaolioon.
        /// </summary>
        /// <param name="obj">Törmäävä olio</param>
        /// <param name="handler">Törmäyksen käsittelevä aliohjelma.</param>
        public void AddCollisionHandler(IPhysicsObject obj, CollisionHandler<IPhysicsObject, IPhysicsObject> ↙
      handler) //ei relevantti

        /// <summary>
        /// Määrää, mihin aliohjelmaan siirrytään kun fysiikkaolio <code>obj</code> törmää johonkin toiseen ↙
    fysiikkaolioon.
        /// </summary>
```

```
        /// <param name="obj">Törmäävä olio</param>
        /// <param name="handler">Törmäyksen käsittelevä aliohjelma.</param>
        public void AddCollisionHandler(PhysicsObject obj, CollisionHandler<PhysicsObject, PhysicsObject> ↙
    handler) //ei relevantti

        /// <summary>
        /// Määrää, mihin aliohjelmaan siirrytään kun fysiikkaolio <code>obj</code> törmää johonkin ↙
    fysiikkarakenteeseen.
        /// </summary>
        /// <param name="obj">Törmäävä olio</param>
        /// <param name="handler">Törmäyksen käsittelevä aliohjelma.</param>
        public void AddCollisionHandler(PhysicsObject obj, CollisionHandler<PhysicsObject, PhysicsStructure> ↙
     handler) //ei relevantti

        /// <summary>
        /// Määrää, mihin aliohjelmaan siirrytään kun fysiikkarakenne <code>o</code> törmää johonkin ↙
    fysiikkaolioon.
        /// </summary>
        /// <param name="obj">Törmäävä fysiikkarakenne</param>
        /// <param name="handler">Törmäyksen käsittelevä aliohjelma.</param>
        public void AddCollisionHandler(PhysicsStructure obj, CollisionHandler<PhysicsStructure, ↙
    PhysicsObject> handler) //ei relevantti

        /// <summary>
        /// Määrää, mihin aliohjelmaan siirrytään kun fysiikkarakenne <code>o</code> törmää toiseen ↙
    fysiikkarakenteeseen.
        /// </summary>
        /// <param name="obj">Törmäävä fysiikkarakenne</param>
        /// <param name="handler">Törmäyksen käsittelevä aliohjelma.</param>
        public void AddCollisionHandler(PhysicsStructure obj, CollisionHandler<PhysicsStructure, ↙
    PhysicsStructure> handler) //ei relevantti

        /// <summary>
        /// Määrää, mihin aliohjelmaan siirrytään kun
        /// olio <code>obj</code> törmää tiettyyn toiseen olioon <code>target</code>.
        /// </summary>
        /// <param name="obj">Törmäävä olio.</param>
        /// <param name="target">Olio johon törmätään.</param>
        /// <param name="handler">Metodi, joka käsittelee törmäyksen (ei parametreja).</param>
        public void AddCollisionHandler<O, T>(O obj, T target, CollisionHandler<PhysicsObject, T> handler)
            where O : IPhysicsObject
            where T : IPhysicsObject //ei relevantti

        /// <summary>
        /// Määrää, mihin aliohjelmaan siirrytään kun
        /// olio <code>obj</code> törmää toiseen olioon, jolla on tietty tagi <code>tag</code>.
        /// </summary>
        /// <param name="obj">Törmäävä olio.</param>
        /// <param name="tag">Törmättävän olion tagi.</param>
        /// <param name="handler">Metodi, joka käsittelee törmäyksen (ei parametreja).</param>
        public void AddCollisionHandler<O, T>(O obj, object tag, CollisionHandler<O, T> handler)
            where O : IPhysicsObject
            where T : IPhysicsObject //ei relevantti

        /// <summary>
        /// Määrää, mihin aliohjelmaan siirrytään kun
        /// yleinen fysiikkaolio <code>obj</code> törmää toiseen yleiseen fysiikkaolioon, jolla on tietty ↙
    tagi <code>tag</code>.
        /// </summary>
        /// <param name="obj">Törmäävä olio.</param>
        /// <param name="tag">Törmättävän olion tagi.</param>
        /// <param name="handler">Metodi, joka käsittelee törmäyksen (ei parametreja).</param>
        public void AddCollisionHandler(IPhysicsObject obj, object tag, CollisionHandler<IPhysicsObject, ↙
    IPhysicsObject> handler) //ei relevantti

        /// <summary>
        /// Määrää, mihin aliohjelmaan siirrytään kun
        /// fysiikkaolio <code>obj</code> törmää toiseen fysiikkaolioon, jolla on tietty tagi <code>tag</ ↙
    code>.
        /// </summary>
        /// <param name="obj">Törmäävä olio.</param>
        /// <param name="tag">Törmättävän olion tagi.</param>
        /// <param name="handler">Metodi, joka käsittelee törmäyksen (ei parametreja).</param>
        public void AddCollisionHandler(PhysicsObject obj, object tag, CollisionHandler<PhysicsObject, ↙
```

```
    PhysicsObject> handler) //ei relevantti

        /// <summary>
        /// Määrää, mihin aliohjelmaan siirrytään kun
        /// fysiikkaolio <code>obj</code> törmää fysiikkarakenteeseen, jolla on tietty tagi <code>tag</code> ↙
    .
        /// </summary>
        /// <param name="obj">Törmäävä olio.</param>
        /// <param name="tag">Törmättävän olion tagi.</param>
        /// <param name="handler">Metodi, joka käsittelee törmäyksen (ei parametreja).</param>
        public void AddCollisionHandler(PhysicsObject obj, object tag, CollisionHandler<PhysicsObject,       ↙
    PhysicsStructure> handler) //ei relevantti

        /// <summary>
        /// Määrää, mihin aliohjelmaan siirrytään kun
        /// fysiikkarakenne <code>obj</code> törmää fysiikkaolioon, jolla on tietty tagi <code>tag</code>.
        /// </summary>
        /// <param name="obj">Törmäävä rakenne.</param>
        /// <param name="tag">Törmättävän olion tagi.</param>
        /// <param name="handler">Metodi, joka käsittelee törmäyksen (ei parametreja).</param>
        public void AddCollisionHandler(PhysicsStructure obj, object tag, CollisionHandler<PhysicsStructure, ↙
     PhysicsObject> handler) //ei relevantti

        /// <summary>
        /// Määrää, mihin aliohjelmaan siirrytään kun
        /// fysiikkarakenne <code>obj</code> törmää toiseen fysiikarakenteeseen, jolla on tietty tagi <code> ↙
    tag</code>.
        /// </summary>
        /// <param name="obj">Törmäävä rakenne.</param>
        /// <param name="tag">Törmättävän rakenteen tagi.</param>
        /// <param name="handler">Metodi, joka käsittelee törmäyksen (ei parametreja).</param>
        public void AddCollisionHandler(PhysicsStructure obj, object tag, CollisionHandler<PhysicsStructure, ↙
     PhysicsStructure> handler) //ei relevantti

        /// <summary>
        /// Poistaa kaikki ehdot täyttävät törmäyksenkäsittelijät.
        /// </summary>
        /// <param name="obj">Törmäävä olio. null jos ei väliä.</param>
        /// <param name="target">Törmäyksen kohde. null jos ei väliä.</param>
        /// <param name="tag">Törmäyksen kohteen tagi. null jos ei väliä.</param>
        /// <param name="handler">Törmäyksenkäsittelijä. null jos ei väliä.</param>
        public void RemoveCollisionHandlers<O, T>(IPhysicsObject obj, IPhysicsObject target, object tag,     ↙
    CollisionHandler<O, T> handler)
            where O : IPhysicsObject
            where T : IPhysicsObject //ei relevantti

        /// <summary>
        /// Poistaa kaikki ehdot täyttävät törmäyksenkäsittelijät.
        /// </summary>
        /// <param name="obj">Törmäävä olio. null jos ei väliä.</param>
        /// <param name="target">Törmäyksen kohde. null jos ei väliä.</param>
        /// <param name="tag">Törmäyksen kohteen tagi. null jos ei väliä.</param>
        /// <param name="handler">Törmäyksenkäsittelijä. null jos ei väliä.</param>
        public void RemoveCollisionHandlers(PhysicsObject obj, PhysicsObject target, object tag,             ↙
    CollisionHandler<PhysicsObject, PhysicsObject> handler) //ei relevantti
    }
}
```

E Farseer Physics -adapterikirjaston PhysicsGame-luokka

```
/*
 * Authors: Tero Jäntti, Tomi Karppinen, Janne Nikkanen.
 */

using System;
using Microsoft.Xna.Framework;
using System.Collections.Generic;
using FarseerPhysics.Common;
using FarseerPhysics.Common.Decomposition;
using FarseerPhysics.Common.PolygonManipulation;
using FarseerPhysics.Factories;
using Jypeli;

namespace AdapteriFarseer
{
    /// <summary>
    /// Peli, jossa on fysiikan laskenta mukana. Peliin lisätyt <code>PhysicsObject</code>-oliot
    /// käyttäytyvät fysiikan lakien mukaan.
    /// </summary>
    public class PhysicsGame : PhysicsGameBase
    {
        private Vector gravity = Vector.Zero;

        /// <summary>
        /// Painovoima. Voimavektori, joka vaikuttaa kaikkiin ei-staattisiin kappaleisiin.
        /// </summary>
        public Vector Gravity
        {
            get
            {
                return gravity;
            }
            set
            {
                gravity = value;
                updatePhysicsConstants();
            }
        }

        public PhysicsLevel Level { get; set; }

        /// <summary>
        /// Alustaa uuden fysiikkapelin.
        /// </summary>
        public PhysicsGame()
            : this( 1 )
        {
        }

        /// <summary>
        /// Alustaa uuden fysiikkapelin.
        /// </summary>
        /// <param name="device">Mikä monitori käytössä, 1=ensimmäinen</param>
        public PhysicsGame(int device)
            : base( device )
        {
            Level = new PhysicsLevel(this);
        }

        private void updatePhysicsConstants()
        {
            if ( gravity != Vector.Zero )
            {
                farseerWorld.Gravity = new Vector2((float)gravity.X, (float)gravity.Y);
            }
        }
    }
}
```

F Farseer Physics -adapterikirjaston PhysicsObject-luokka

```
/*
 * Authors: Tero Jäntti, Tomi Karppinen, Janne Nikkanen.
 */

using System;
using System.Diagnostics;
using Microsoft.Xna.Framework;
using System.Collections.Generic;
using Jypeli;
using FarseerPhysics.Dynamics;
using AdvanceMath;

using FarseerPhysics.Common;
using FarseerPhysics.Common.Decomposition;
using FarseerPhysics.Common.PolygonManipulation;

#if !XBOX

#endif

namespace AdapteriFarseer
{
    [Obsolete("Use newer collision handlers with IPhysicsObject for structured object support")]
    public delegate void OldCollisionHandler(PhysicsObject collidingObject, PhysicsObject otherObject);


    /// <summary>
    /// Törmäyskuvion laatuun vaikuttavat parametrit.
    /// </summary>
    public struct CollisionShapeParameters //ei relevantti


    /// <summary>
    /// Kappaleen kuvion laatu törmäyksentunnistuksessa.
    /// </summary>
    [Obsolete("Use CollisionShapeParameters or the PhysicsTemplates class.")]
    public struct CollisionShapeQuality //ei relevantti

    internal class Force //ei relevantti


    /// <summary>
    /// Peliolio, joka noudattaa fysiikkamoottorin määräämiä fysiikan lakeja.
    /// Voidaan kuitenkin myös laittaa noudattamaan lakeja valikoidusti.
    /// </summary>
    public class PhysicsObject : GameObject, IPhysicsObject
    {
        private const double DefaultMass = 1.0;

        /// <summary>
        /// Olioon vaikuttavat voimat
        /// </summary>
        internal List<Force> ActiveForces = new List<Force>();

        public Body Body { get; private set; }

        public int CollisionIgnoreGroup;

        private bool _ignoresCollisionResponse;

        /// <summary>
        /// Rakenneolio, johon tämä olio kuuluu.
        /// </summary>
        public PhysicsStructure ParentStructure { get; internal set; }

        /// <summary>
        /// Olion paikka koordinaatistossa. Käsittää sekä X- että Y-koordinaatin.
        /// </summary>
        [Save]
        public override Vector Position
        {
            get
```

```
            {
                return new Vector(Body.Position.X, Body.Position.Y);
            }
            set
            {
                Body.Position = new Vector2((float)value.X, (float)value.Y);
            }
        }

        /// <summary>
        /// Olion koko (x on leveys, y on korkeus).
        /// </summary>
        [Save]
        public override Vector Size
        {
            get
            {
                return base.Size;
            }
            set
            {
                base.Size = value;
            }
        }

        /// <summary>
        /// Kulma, jossa olio on. Oliota voi pyörittää kulmaa vaihtamalla.
        /// </summary>
        [Save]
        public override Angle Angle
        {
            get
            {
                return Angle.FromRadians(Body.Rotation);
            }
            set
            {
                Body.Rotation = (float)value.Radians;
            }
        }

        /// <summary>
        /// Jos <c>false</c>, olio ei voi pyöriä.
        /// </summary>
        public bool CanRotate
        {
            get { return !float.IsPositiveInfinity(Body.Inertia); }
            set
            {
                if (!value && Body.BodyType == FarseerPhysics.Dynamics.BodyType.Dynamic)
                {
                    Body.Inertia = float.PositiveInfinity;
                }
                else
                {
                    UpdateBody( this.Mass );
                }
            }
        }

        /// <summary>
        /// Olion massa. Mitä suurempi massa, sitä suurempi voima tarvitaan olion liikuttamiseksi.
        /// </summary>
        /// <remarks>
        /// Massan asettaminen muuttaa myös hitausmomenttia (<c>MomentOfInertia</c>).
        /// </remarks>
        public double Mass
        {
            get { return Body.Mass; }
            set
            {
                // We should change the moment of intertia as well. If the mass is changed like, from
                // 1.0 to 100000.0, it would look funny if a heavy object would spin wildly from a small
                // touch.
```

```
                if ( !CanRotate )
                {
                    // The moment of inertia has been set to positive infinity,
                    // let's keep it that way and just set the mass.
                    Body.Mass = (float)value;
                }
                else
                {
                    UpdateBody( value );
                }
            }
        }

        /// <summary>
        /// Olion hitausmomentti. Mitä suurempi hitausmomentti, sitä enemmän vääntöä tarvitaan
        /// olion pyörittämiseksi.
        /// </summary>
        public double MomentOfInertia
        {
            get { return Body.Inertia; }
            set { Body.Inertia = (float)value; }
        }

        /// <summary>
        /// Lepokitka. Liikkeen alkamista vastustava voima, joka ilmenee kun olio yrittää lähteä liikkeelle
        /// toisen olion pinnalta (esim. laatikkoa yritetään työntää eteenpäin).
        /// </summary>
        public double StaticFriction { get; set; }

        /// <summary>
        /// Liikekitka. Liikettä vastustava voima joka ilmenee kun kaksi oliota liikkuu toisiaan vasten
        /// (esim. laatikko liukuu maata pitkin). Arvot välillä 0.0 (ei kitkaa) ja 1.0 (täysi kitka).
        /// </summary>
        public double KineticFriction
        {
            get { return Body.Friction; }
            set { Body.Friction = (float)value; }
        }

        /// <summary>
        /// Olion kimmoisuus. Arvo välillä 0.0-1.0.
        /// Arvolla 1.0 olio säilyttää kaiken vauhtinsa törmäyksessä. Mitä pienempi arvo,
        /// sitä enemmän olion vauhti hidastuu törmäyksessä.
        /// </summary>
        public double Restitution
        {
            get { return Body.Restitution; }
            set { Body.Restitution = (float)value; }
        }

        /// <summary>
        /// Olion nopeus.
        /// </summary>
        [Save]
        public Vector Velocity
        {
            get
            {
                return new Vector(Body.LinearVelocity.X, Body.LinearVelocity.Y);
            }
            set
            {
                Body.LinearVelocity = new Vector2((float)value.X, (float)value.Y);
            }
        }

        /// <summary>
        /// Olion kulmanopeus.
        /// </summary>
        [Save]
        public double AngularVelocity
        {
            get
```

```
            {
                return Body.AngularVelocity;
            }
            set
            {
                Body.AngularVelocity = (float)value;
            }
        }

        public Vector Acceleration { get; set; }

        public double AngularAcceleration { get; set; }

        /// <summary>
        /// Olion muoto.
        /// </summary>
        public override Shape Shape
        {
            get { return base.Shape; }
            set
            {
                SetShape( value, GetDefaultParameters( Width, Height ) );
            }
        }

        internal void SetShape( Shape shape, CollisionShapeParameters parameters )
        {
            base.Shape = shape;
            IShape physicsShape = CreatePhysicsShape(shape, Size, parameters);
            Body uusiBody = null;
            uusiBody = luoFarseerBody(physicsShape);
            PhysicsGame.farseerWorld.RemoveBody(Body);
            uusiBody.UserData = Body.UserData;
            uusiBody.BodyType = Body.BodyType;
            uusiBody.Mass = Body.Mass;
            uusiBody.Restitution = Body.Restitution;
            uusiBody.Friction = Body.Friction;
            uusiBody.OnCollision += tormaysTapahtuma;
            Body = uusiBody;
        }

        /// <summary>
        /// Olion hidastuminen. Hidastaa olion vauhtia, vaikka se ei
        /// osuisi mihinkään. Vähän kuin väliaineen (esim. ilman tai veden)
        /// vastus. Oletusarvo on 1.0, jolloin hidastumista ei ole. Mitä
        /// pienempi arvo, sitä enemmän kappale hidastuu.
        ///
        /// Yleensä kannattaa käyttää arvoja, jotka ovat lähellä ykköstä,
        /// esim. 0.95.
        /// </summary>
        public double LinearDamping
        {
            get { return Body.LinearDamping; }
            set { Body.LinearDamping = (float)value; }
        }

        /// <summary>
        /// Olion pyörimisen hidastuminen.
        ///
        /// <see cref="LinearDamping"/>
        /// </summary>
        public double AngularDamping
        {
            get { return Body.AngularDamping; }
            set { Body.AngularDamping = (float)value; }
        }

        /// <summary>
        /// Jättääkö olio räjähdyksen paineaallon huomiotta.
        /// </summary>
        public bool IgnoresExplosions { get; set; }

        /// <summary>
        /// Jättääkö olio törmäyksen huomioimatta. Jos tosi, törmäyksestä
```

```
        /// tulee tapahtuma, mutta itse törmäystä ei tapahdu.
        /// </summary>
        public bool IgnoresCollisionResponse
        {
            get
            {
                return _ignoresCollisionResponse;
            }
            set
            {
                _ignoresCollisionResponse = value;
                Body.IsSensor = value;
            }
        }

        /// <summary>
        /// Jättääkö olio painovoiman huomioimatta.
        /// </summary>
        public bool IgnoresGravity
        {
            get { return Body.IgnoreGravity; }
            set { Body.IgnoreGravity = value; }
        }

        /// <summary>
        /// Jättääkö olio kaikki fysiikkalogiikat (ks. <c>AddPhysicsLogic</c>)
        /// huomiotta. Vaikuttaa esim. painovoimaan, mutta ei törmäyksiin.
        /// </summary>
        public bool IgnoresPhysicsLogics
        {
            get { return !Body.Awake; }
            set { Body.Awake = !value; }
        }

        /// <summary>
        /// Tapahtuu kun olio törmää toiseen.
        /// </summary>
        public event CollisionHandler<IPhysicsObject, IPhysicsObject> Collided;

        private void OnCollided( object sender, CollisionEventArgs args )
        {
            if ( Collided == null ||this.IsDestroyed || args.Other == null ) return;
            var other = (PhysicsObject)args.Other.UserData;
            if ( other.IsDestroyed ) return;

            Collided( this, other );
            Brain.OnCollision( other );
        }

        /// <summary>
        /// Tekee suorakulmaisen kappaleen, joka on staattinen (eli pysyy paikallaan).
        /// </summary>
        /// <param name="width">Kappaleen leveys</param>
        /// <param name="height">Kappaleen korkeus</param>
        /// <returns>Fysiikkaolio</returns>
        public static PhysicsObject CreateStaticObject( double width, double height ) //ei relevantti

        /// <summary>
        /// Tekee suorakulmaisen kappaleen, joka on staattinen (eli pysyy paikallaan).
        /// Kappaleen koko ja ulkonäkö ladataan parametrina annetusta kuvasta.
        /// </summary>
        /// <param name="width">Kappaleen leveys</param>
        /// <param name="height">Kappaleen korkeus</param>
        /// <returns>Fysiikkaolio</returns>
        public static PhysicsObject CreateStaticObject( Image image ) //ei relevantti

        /// <summary>
        /// Tekee vapaamuotoisen kappaleen, joka on staattinen (eli pysyy paikallaan).
        /// </summary>
        /// <param name="width">Kappaleen leveys</param>
        /// <param name="height">Kappaleen korkeus</param>
        /// <param name="shape">Kappaleen muoto</param>
        /// <returns>Fysiikkaolio</returns>
        public static PhysicsObject CreateStaticObject( double width, double height, Shape shape ) //ei ↙
```

```
relevantti

    [Obsolete( "Use CollisionShapeParameters or the PhysicsTemplates class." )]
    public static PhysicsObject CreateStaticObject( double width, double height, Shape shape, 
CollisionShapeQuality quality ) //ei relevantti

    public static PhysicsObject CreateStaticObject( double width, double height, Shape shape, 
CollisionShapeParameters parameters ) //ei relevantti

    [Obsolete("Use function with CollisionShapeParameters")]
    public static PhysicsObject CreateStaticObject( double width, double height, Shape shape, double 
maxDistanceBetweenVertices, double gridSpacing ) //ei relevantti

    internal static PhysicsObject CreateStaticObject( double width, double height, Shape shape, IShape 
physicsShape ) //ei relevantti

    /// <summary>
    /// Luo uuden fysiikkaolion.
    /// </summary>
    /// <param name="width">Leveys.</param>
    /// <param name="height">Korkeus.</param>
    public PhysicsObject( double width, double height )
        : this( width, height, Shape.Rectangle )
    {
    }

    /// <summary>
    /// Luo uuden fysiikkaolion.
    /// </summary>
    /// <param name="width">Leveys.</param>
    /// <param name="height">Korkeus.</param>
    /// <param name="shape">Muoto.</param>
    public PhysicsObject( double width, double height, Shape shape )
        : this( width, height, shape, CreatePhysicsShape( shape, new Vector( width, height ) ) )
    {
    }

    /// <summary>
    /// Luo uuden fysiikkaolion.
    /// Kappaleen koko ja ulkonäkö ladataan parametrina annetusta kuvasta.
    /// </summary>
    /// <param name="image">Kuva</param>
    public PhysicsObject( Image image )
        : this( image.Width, image.Height, Shape.Rectangle )
    {
        this.Image = image;
    }

    public PhysicsObject( double width, double height, Shape shape, CollisionShapeParameters 
shapeParameters )
        : this( width, height, shape, CreatePhysicsShape( shape, new Vector( width, height ), 
shapeParameters ) )
    {
    }

    [Obsolete( "Use CollisionShapeParameters or the PhysicsTemplates class." )]
    public PhysicsObject( double width, double height, Shape shape, CollisionShapeQuality quality )
        : this( width, height, shape, CreatePhysicsShape( shape, new Vector( width, height ) ) )
    {
    }

    [Obsolete( "Use constructor with CollisionShapeParameters" )]
    public PhysicsObject( double width, double height, Shape shape, double maxDistanceBetweenVertices, 
double gridSpacing )
        : this( width, height, shape, new CollisionShapeParameters( maxDistanceBetweenVertices, 
gridSpacing ) )
    {
    }

    /// <summary>
    /// Luo fysiikkaolion, jonka muotona on säde.
    /// </summary>
    /// <param name="raySegment">Säde.</param>
    public PhysicsObject( RaySegment raySegment )
```

```
            : this( 1, 1, raySegment )
        {
        }

        /// <summary>
        /// Initializes the object with the given physics shape. The size of
        /// the physicsShape must be the one given.
        /// </summary>
        internal PhysicsObject(double width, double height, Shape shape, IShape physicsShape)
            : base( width, height, shape )
        {
            Body = luoFarseerBody(physicsShape);
            Body.BodyType = FarseerPhysics.Dynamics.BodyType.Dynamic;
            Body.UserData = this;
            Body.Mass = 1.0f;
            Body.Restitution = 0.5f;
            Body.Friction = 0.4f;
            Body.OnCollision += tormaysTapahtuma;
        }

        private Body luoFarseerBody(IShape physicsShape)
        {
            Body body = null;
            if (physicsShape is CircleShape)
            {
                CircleShape circleShape = (CircleShape)physicsShape;
                FarseerPhysics.Collision.Shapes.CircleShape farseerCircleShape = new FarseerPhysics.      ↙
    Collision.Shapes.CircleShape((float)circleShape.Radius, 1f);
                body = FarseerPhysics.Factories.BodyFactory.CreateCircle(PhysicsGame.farseerWorld, (float)   ↙
    circleShape.Radius, 1f);
            }
            else if (physicsShape is PolygonShape && this.Shape.Cache != null)
            {
                FarseerPhysics.Collision.Shapes.PolygonShape farseerPolygonShape = new FarseerPhysics.      ↙
    Collision.Shapes.PolygonShape(1f);
                Vector2D[] vertexes = physicsShape.Vertexes;
                Vector2[] lista = new Vector2[vertexes.Length];
                for (int i = 0; i < vertexes.Length; i++)
                {
                    lista[i] = new Vector2((float)vertexes[i].X, (float)vertexes[i].Y);
                }
                FarseerPhysics.Common.Vertices vertices = new FarseerPhysics.Common.Vertices(lista);
                if (this.Shape is Jypeli.Rectangle)
                    body = FarseerPhysics.Factories.BodyFactory.CreatePolygon(PhysicsGame.farseerWorld,     ↙
    vertices, 1f);
                else
                    body = FarseerPhysics.Factories.BodyFactory.CreateLoopShape(PhysicsGame.farseerWorld,   ↙
    vertices);
            }
            else if (physicsShape is PolygonShape && this.Shape is Polygon && this.Shape.Cache == null)
            {
                Polygon polygon = (Polygon)this.Shape;
                uint[] data = new uint[polygon.Texture.Width * polygon.Texture.Height];
                polygon.Texture.GetData(data);
                Vertices textureVertices = PolygonTools.CreatePolygon(data, polygon.Texture.Width, false);
                Vector2 centroid = -textureVertices.GetCentroid();
                textureVertices.Translate(ref centroid);
                textureVertices = SimplifyTools.ReduceByDistance(textureVertices, 4f);
                List<Vertices> list = BayazitDecomposer.ConvexPartition(textureVertices);
                body = FarseerPhysics.Factories.BodyFactory.CreateCompoundPolygon(PhysicsGame.farseerWorld,  ↙
    list, 1f);
            }
            else body = Body;
            return body;
        }

        /// <summary>
        ///
        /// </summary>
        /// <param name="adapteeCollider"></param>
        /// <param name="adapteeCollidee"></param>
        /// <param name="contact"></param>
        /// <returns></returns>
        public bool tormaysTapahtuma(FarseerPhysics.Dynamics.Fixture adapteeCollider, FarseerPhysics.      ↙
```

```
    Dynamics.Fixture adapteeCollidee, FarseerPhysics.Dynamics.Contacts.Contact contact)
        {
            OnCollided(adapteeCollider.Body, new CollisionEventArgs(0, adapteeCollidee.Body));
            return true;
        }

        private static CollisionShapeParameters GetDefaultParameters( double width, double height )
        {
            CollisionShapeParameters p;
            p.MaxVertexDistance = Math.Min( width, height ) / 3;
            p.DistanceGridSpacing = Math.Min( width, height ) / 2;
            return p;
        }

        /// <summary>
        /// Tekee oliosta staattisen. Staattinen olio ei liiku muiden olioiden törmäyksistä,
        /// vaan ainoastaan muuttamalla suoraan sen paikkaa tai nopeutta.
        /// </summary>
        public void MakeStatic()
        {
            Body.BodyType = FarseerPhysics.Dynamics.BodyType.Kinematic;
        }

        /// <summary>
        /// This updates the mass and momentOfInertia based on the new mass.
        /// </summary>
        private void UpdateBody( double mass )
        {
            Body.Mass = (float)mass;
        }

        internal static IShape CreatePhysicsShape( Shape shape, Vector size )
        {
            return CreatePhysicsShape( shape, size, GetDefaultParameters( size.X, size.Y ) );
        }

        /// <summary>
        /// Creates a shape to be used in the Physics Body. A physics shape is scaled to the
        /// size of the object. In addition, it has more vertices and some additional info
        /// that is used in collision detection.
        /// </summary>
        internal static IShape CreatePhysicsShape( Shape shape, Vector size, CollisionShapeParameters     ↙
    parameters )
        {
            if ( shape is RaySegment )
            {
                RaySegment raySegment = (RaySegment)shape;
                AdapteriRaySegment singleSegment = new AdapteriRaySegment(
                    new Vector2D(raySegment.Origin.X, raySegment.Origin.Y),
                    new Vector2D(raySegment.Direction.X, raySegment.Direction.Y),
                    raySegment.Length );
                return new RaySegmentsShape( singleSegment );
            }
            else if ( shape is Ellipse )
            {
                Debug.Assert( shape.IsUnitSize );

                double smaller = Math.Min( size.X, size.Y );
                double bigger = Math.Max( size.X, size.Y );
                // Average between width and height.
                double r = smaller / 2 + ( bigger - smaller ) / 2;
                int vertexCount = (int)Math.Ceiling( ( 2 * Math.PI * r ) / parameters.MaxVertexDistance );

                if ( Math.Abs(size.X - size.Y) <= double.Epsilon )
                {
                    // We get more accurate results by using the circleshape.
                    // in addition, the circleshape does not need a DistanceGrid
                    // object (which is slow to initialize) because calculations
                    // for a circleshape are much simpler.
                    return new CircleShape( r, vertexCount );
                }
                else
                {
                    Vector2D[] vertexes = new Vector2D[vertexCount];
```

```
                double a = 0.5 * size.X;
                double b = 0.5 * size.Y;

                for ( int i = 0; i < vertexCount; i++ )
                {
                    double t = ( i * 2 * Math.PI ) / vertexCount;
                    double x = a * Math.Cos( t );
                    double y = b * Math.Sin( t );
                    vertexes[i] = new Vector2D( x, y );
                }

                return new PolygonShape( vertexes, parameters.DistanceGridSpacing );
            }
        }
        else
        {
            if (shape.Cache != null)
            {
                Vector2D[] originalVertexes = new Vector2D[shape.Cache.OutlineVertices.Length];
                for (int i = 0; i < shape.Cache.OutlineVertices.Length; i++)
                {
                    Vector v = shape.Cache.OutlineVertices[i];
                    if (shape.IsUnitSize)
                    {
                        v.X *= size.X;
                        v.Y *= size.Y;
                    }
                    originalVertexes[i] = new Vector2D(v.X, v.Y);
                }

                Vector2D[] polyVertexes = VertexHelper.Subdivide(originalVertexes, parameters. ↙
MaxVertexDistance);

                return new PolygonShape(polyVertexes, parameters.DistanceGridSpacing);
            }
            else return new PolygonShape(null, 0);
        }
    }

    /// <summary>
    /// Työntää oliota.
    /// </summary>
    /// <param name="force">Voima, jolla oliota työnnetään.</param>
    public virtual void Push( Vector force )
    {
        Body.ApplyForce(new Vector2((float)force.X, (float)force.Y));
    }

    /// <summary>
    /// Työntää oliota tietyn ajan tietyllä voimalla.
    /// </summary>
    /// <param name="force">Voima, jolla oliota työnnetään sekunnissa.</param>
    /// <param name="time">Aika, kuinka kauan voimaa pidetään yllä.</param>
    public virtual void Push( Vector force, TimeSpan time )
    {
        Body.ApplyForce(new Vector2((float)force.X, (float)force.Y));
        ActiveForces.Add( new Force( force, time ) );
    }

    /// <summary>
    /// Kohdistaa kappaleeseen impulssin. Tällä kappaleen saa nopeasti liikkeeseen.
    /// </summary>
    public virtual void Hit( Vector impulse )
    {
        Body.ApplyLinearImpulse(new Vector2((float)impulse.X, (float)impulse.Y));
    }

    /// <summary>
    /// Kohdistaa kappaleeseen vääntövoiman. Voiman suunta riippuu merkistä.
    /// </summary>
    /// <param name="torque">Vääntövoima.</param>
    public virtual void ApplyTorque( double torque )
    {
```

```
            Body.ApplyTorque((float)torque);
        }

        /// <summary>
        /// Pysäyttää olion.
        /// </summary>
        public virtual void Stop()
        {
            Body.ApplyForce(Vector2.Zero);
            Body.LinearVelocity = Vector2.Zero;
            Body.AngularVelocity = 0.0f;
        }

        /// <summary>
        /// Pysäyttää olion liikkeen vaakasuunnassa.
        /// </summary>
        public void StopHorizontal()
        {
            Vector2 oldVel = Body.LinearVelocity;
            Body.LinearVelocity = new Vector2(0.0f, oldVel.Y);
        }

        /// <summary>
        /// Pysäyttää olion liikkeen pystysuunnassa.
        /// </summary>
        public void StopVertical()
        {
            Vector2 oldVel = Body.LinearVelocity;
            Body.LinearVelocity = new Vector2(oldVel.X, 0.0f);
        }

        public override void Destroy()
        {
            base.Destroy();
        }

        public override void Update( Time time )
        {
            for ( int i = ActiveForces.Count - 1; i >= 0; i-- )
            {
                if ( ActiveForces[i].IsDestroyed() )
                {
                    ActiveForces.RemoveAt( i );
                    continue;
                }

                // Apply the force
                Push( ActiveForces[i].Value * time.SinceLastUpdate.TotalSeconds );
            }

            base.Update( time );
        }

        /// <summary>
        /// Siirtää oliota.
        /// </summary>
        /// <param name="movement">Vektori, joka määrittää kuinka paljon siirretään.</param>
        public override void Move( Vector movement )
        {
            Vector dv = movement - this.Velocity;
            Hit( Mass * dv );
        }

        protected override void MoveToTarget()
        {
            if ( !moveTarget.HasValue )
            {
                Stop();
                moveTimer.Stop();
                return;
            }

            Vector d = moveTarget.Value - Position;
            double vt = moveSpeed * moveTimer.Interval;
```

```
            if ( d.Magnitude < vt )
            {
                Vector targetLoc = moveTarget.Value;
                Stop();
                moveTimer.Stop();
                moveTarget = null;
                OnArrived( targetLoc );
            }
            else
            {
                Vector dv = Vector.FromLengthAndAngle( moveSpeed, d.Angle ) - this.Velocity;
                Hit( Mass * dv );
            }
        }
    }
}
```

G Physics2DDotNet-adapterikirjaston PhysicsGameBase-luokka

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using Physics2DDotNet;
using Physics2DDotNet.Joints;
using Jypeli;

namespace AdapteriPhysics2D
{
    /// <summary>
    /// Kantaluokka fysiikkapeleille.
    /// </summary>
    public abstract class PhysicsGameBase : Game
    {
        protected struct CollisionRecord // ei relevantti

        protected PhysicsEngine phsEngine;
        private SynchronousList<Joint> Joints = new SynchronousList<Joint>();
        protected Dictionary<CollisionRecord, CollisionHandler<IPhysicsObject, IPhysicsObject>>  ↙
    collisionHandlers =
            new Dictionary<CollisionRecord, CollisionHandler<IPhysicsObject, IPhysicsObject>>();

        /// <summary>
        /// Onko fysiikan laskenta käytössä vai ei.
        /// </summary>
        public bool PhysicsEnabled { get; set; }

        /// <summary>
        /// Alustaa uuden fysiikkapelin.
        /// </summary>
        /// <param name="device">Mikä monitori käytössä, 1=ensimmäinen</param>
        public PhysicsGameBase( int device )
            : base( device )
        {
            PhysicsEnabled = true;
            phsEngine = new PhysicsEngine();

            Joints.ItemAdded += OnJointAdded;
            Joints.ItemRemoved += OnJointRemoved;
        }

        void OnJointAdded( Joint j )
        {
            j.Lifetime.IsExpired = false;
            phsEngine.AddJoint( j );
        }

        void OnJointRemoved( Joint j )
        {
            j.Lifetime.IsExpired = true;
        }

        protected override void OnObjectAdded( IGameObject obj )
        {
            if ( obj is PhysicsObject )
            {
                AddToEngine( (PhysicsObject)obj );
            }

            base.OnObjectAdded( obj );
        }

        protected override void OnObjectRemoved( IGameObject obj )
        {
            if ( obj is PhysicsObject )
            {
                RemoveFromEngine( (PhysicsObject)obj );
            }

            base.OnObjectRemoved( obj );
        }

        /// <summary>
```

```
        /// Pysäyttää kaiken liikkeen.
        /// </summary>
        public void StopAll()
        {
            foreach ( var layer in Layers )
            {
                foreach ( var obj in layer.Objects )
                {
                    if ( obj is PhysicsObject )
                        ( (PhysicsObject)obj ).Stop();
                }
            }
        }

        /// <summary>
        /// Nollaa kaiken (kontrollit, näyttöobjektit, ajastimet ja fysiikkamoottorin).
        /// </summary>
        public override void ClearAll()
        {
            ClearPhysics();
            base.ClearAll();
        }

        /// <summary>
        /// Nollaa fysiikkamoottorin.
        /// </summary>
        private void ClearPhysics()
        {
            phsEngine.Clear();
        }

        private void AddToEngine( PhysicsObject po )
        {
            if ( po.Body.Engine != null )
                return;

            po.Body.Lifetime.IsExpired = false;
            phsEngine.AddBody( po.Body );
        }

        private void RemoveFromEngine( PhysicsObject po )
        {
            if ( po.Body.Engine == null )
                return;

            po.Body.Lifetime.IsExpired = true;
        }

        /// <summary>
        /// Lisää liitoksen peliin.
        /// </summary>
        public void Add( Physics2DDotNet.Joints.Joint j )
        {
            Joints.Add( j );
        }

        /// <summary>
        /// Poistaa liitoksen pelistä.
        /// </summary>
        /// <param name="j"></param>
        internal void Remove( Physics2DDotNet.Joints.Joint j )
        {
            Joints.Remove( j );
        }

        /// <summary>
        /// Poistaa liitoksen pelistä.
        /// </summary>
        /// <param name="j"></param>
        internal void Remove(AxleJoint j)
        {
            Joints.Remove( j.innerJoint );
        }
```

```
        /// <summary>
        /// Lisää liitoksen peliin.
        /// </summary>
        public void Add(AxleJoint j)
        {
            if ( !j.Object1.IsAddedToGame ) Add( j.Object1 );
            if ( j.Object2 != null && !j.Object2.IsAddedToGame ) Add( j.Object2 );

            Add( j.innerJoint );
        }

        public override void Add( IGameObject o, int layer )
        {
            if ( o is PhysicsStructure )
            {
                foreach ( var joint in ( (PhysicsStructure)o ).Joints )
                {
                    Add( joint );
                }
            }

            base.Add( o, layer );
        }

        /// <summary>
        /// Ajetaan kun pelin tilannetta päivitetään. Päivittämisen voi toteuttaa perityssä luokassa
        /// toteuttamalla tämän metodin. Perityn luokan metodissa tulee kutsua kantaluokan metodia.
        /// </summary>
        /// <param name="time"></param>
        protected override void Update( Time time )
        {
            double dt = time.SinceLastUpdate.TotalSeconds;

            if ( PhysicsEnabled )
            {
                phsEngine.Update( dt );
            }

            base.Update( time );

            // Updating joints must be after base.Update so that the bodies
            // are added to the engine before the joints
            Joints.Update( time );
        }

        /// <summary>
        /// Määrää, mihin aliohjelmaan siirrytään kun olio <code>obj</code> törmää johonkin toiseen olioon.
        /// </summary>
        /// <typeparam name="T">Kohdeolion tyyppi.</typeparam>
        /// <param name="obj">Törmäävä olio</param>
        /// <param name="handler">Törmäyksen käsittelevä aliohjelma.</param>
        public void AddCollisionHandler<O, T>( O obj, CollisionHandler<O, T> handler )
            where O : IPhysicsObject
            where T : IPhysicsObject // ei relevantti

        /// <summary>
        /// Määrää, mihin aliohjelmaan siirrytään kun yleinen fysiikkaolio <code>obj</code>
        /// törmää johonkin toiseen yleiseen fysiikkaolioon.
        /// </summary>
        /// <param name="obj">Törmäävä olio</param>
        /// <param name="handler">Törmäyksen käsittelevä aliohjelma.</param>
        public void AddCollisionHandler( IPhysicsObject obj, CollisionHandler<IPhysicsObject,              ↙
    IPhysicsObject> handler ) // ei relevantti

        /// <summary>
        /// Määrää, mihin aliohjelmaan siirrytään kun fysiikkaolio <code>obj</code> törmää johonkin toiseen ↙
    fysiikkaolioon.
        /// </summary>
        /// <param name="obj">Törmäävä olio</param>
        /// <param name="handler">Törmäyksen käsittelevä aliohjelma.</param>
        public void AddCollisionHandler( PhysicsObject obj, CollisionHandler<PhysicsObject, PhysicsObject>  ↙
    handler ) // ei relevantti

        /// <summary>
```

```
        /// Määrää, mihin aliohjelmaan siirrytään kun fysiikkaolio <code>obj</code> törmää johonkin ↙
    fysiikkarakenteeseen.
        /// </summary>
        /// <param name="obj">Törmäävä olio</param>
        /// <param name="handler">Törmäyksen käsittelevä aliohjelma.</param>
        public void AddCollisionHandler( PhysicsObject obj, CollisionHandler<PhysicsObject, ↙
    PhysicsStructure> handler ) // ei relevantti

        /// <summary>
        /// Määrää, mihin aliohjelmaan siirrytään kun fysiikkarakenne <code>o</code> törmää johonkin ↙
    fysiikkaolioon.
        /// </summary>
        /// <param name="obj">Törmäävä fysiikkarakenne</param>
        /// <param name="handler">Törmäyksen käsittelevä aliohjelma.</param>
        public void AddCollisionHandler( PhysicsStructure obj, CollisionHandler<PhysicsStructure, ↙
    PhysicsObject> handler ) // ei relevantti

        /// <summary>
        /// Määrää, mihin aliohjelmaan siirrytään kun fysiikkarakenne <code>o</code> törmää toiseen ↙
    fysiikkarakenteeseen.
        /// </summary>
        /// <param name="obj">Törmäävä fysiikkarakenne</param>
        /// <param name="handler">Törmäyksen käsittelevä aliohjelma.</param>
        public void AddCollisionHandler( PhysicsStructure obj, CollisionHandler<PhysicsStructure, ↙
    PhysicsStructure> handler ) // ei relevantti

        /// <summary>
        /// Määrää, mihin aliohjelmaan siirrytään kun
        /// olio <code>obj</code> törmää tiettyyn toiseen olioon <code>target</code>.
        /// </summary>
        /// <param name="obj">Törmäävä olio.</param>
        /// <param name="target">Olio johon törmätään.</param>
        /// <param name="handler">Metodi, joka käsittelee törmäyksen (ei parametreja).</param>
        public void AddCollisionHandler<O, T>( O obj, T target, CollisionHandler<PhysicsObject, T> handler )
            where O : IPhysicsObject
            where T : IPhysicsObject // ei relevantti

        /// <summary>
        /// Määrää, mihin aliohjelmaan siirrytään kun
        /// olio <code>obj</code> törmää toiseen olioon, jolla on tietty tagi <code>tag</code>.
        /// </summary>
        /// <param name="obj">Törmäävä olio.</param>
        /// <param name="tag">Törmättävän olion tagi.</param>
        /// <param name="handler">Metodi, joka käsittelee törmäyksen (ei parametreja).</param>
        public void AddCollisionHandler<O, T>( O obj, object tag, CollisionHandler<O, T> handler )
            where O : IPhysicsObject
            where T : IPhysicsObject // ei relevantti

        /// <summary>
        /// Määrää, mihin aliohjelmaan siirrytään kun
        /// yleinen fysiikkaolio <code>obj</code> törmää toiseen yleiseen fysiikkaolioon, jolla on tietty ↙
    tagi <code>tag</code>.
        /// </summary>
        /// <param name="obj">Törmäävä olio.</param>
        /// <param name="tag">Törmättävän olion tagi.</param>
        /// <param name="handler">Metodi, joka käsittelee törmäyksen (ei parametreja).</param>
        public void AddCollisionHandler( IPhysicsObject obj, object tag, CollisionHandler<IPhysicsObject, ↙
    IPhysicsObject> handler ) // ei relevantti

        /// <summary>
        /// Määrää, mihin aliohjelmaan siirrytään kun
        /// fysiikkaolio <code>obj</code> törmää toiseen fysiikkaolioon, jolla on tietty tagi <code>tag</ ↙
    code>.
        /// </summary>
        /// <param name="obj">Törmäävä olio.</param>
        /// <param name="tag">Törmättävän olion tagi.</param>
        /// <param name="handler">Metodi, joka käsittelee törmäyksen (ei parametreja).</param>
        public void AddCollisionHandler( PhysicsObject obj, object tag, CollisionHandler<PhysicsObject, ↙
    PhysicsObject> handler ) // ei relevantti

        /// <summary>
        /// Määrää, mihin aliohjelmaan siirrytään kun
        /// fysiikkaolio <code>obj</code> törmää fysiikkarakenteeseen, jolla on tietty tagi <code>tag</code> ↙
    .
```

```
        /// </summary>
        /// <param name="obj">Törmäävä olio.</param>
        /// <param name="tag">Törmättävän olion tagi.</param>
        /// <param name="handler">Metodi, joka käsittelee törmäyksen (ei parametreja).</param>
        public void AddCollisionHandler( PhysicsObject obj, object tag, CollisionHandler<PhysicsObject, ↙
    PhysicsStructure> handler ) // ei relevantti

        /// <summary>
        /// Määrää, mihin aliohjelmaan siirrytään kun
        /// fysiikkarakenne <code>obj</code> törmää fysiikkaolioon, jolla on tietty tagi <code>tag</code>.
        /// </summary>
        /// <param name="obj">Törmäävä rakenne.</param>
        /// <param name="tag">Törmättävän olion tagi.</param>
        /// <param name="handler">Metodi, joka käsittelee törmäyksen (ei parametreja).</param>
        public void AddCollisionHandler( PhysicsStructure obj, object tag, CollisionHandler<PhysicsStructure ↙
    , PhysicsObject> handler ) // ei relevantti

        /// <summary>
        /// Määrää, mihin aliohjelmaan siirrytään kun
        /// fysiikkarakenne <code>obj</code> törmää toiseen fysiikarakenteeseen, jolla on tietty tagi <code> ↙
    tag</code>.
        /// </summary>
        /// <param name="obj">Törmäävä rakenne.</param>
        /// <param name="tag">Törmättävän rakenteen tagi.</param>
        /// <param name="handler">Metodi, joka käsittelee törmäyksen (ei parametreja).</param>
        public void AddCollisionHandler( PhysicsStructure obj, object tag, CollisionHandler<PhysicsStructure ↙
    , PhysicsStructure> handler ) // ei relevantti

        /// <summary>
        /// Poistaa kaikki ehdot täyttävät törmäyksenkäsittelijät.
        /// </summary>
        /// <param name="obj">Törmäävä olio. null jos ei väliä.</param>
        /// <param name="target">Törmäyksen kohde. null jos ei väliä.</param>
        /// <param name="tag">Törmäyksen kohteen tagi. null jos ei väliä.</param>
        /// <param name="handler">Törmäyksenkäsittelijä. null jos ei väliä.</param>
        public void RemoveCollisionHandlers<O,T>( IPhysicsObject obj, IPhysicsObject target, object tag, ↙
    CollisionHandler<O,T> handler )
            where O : IPhysicsObject
            where T : IPhysicsObject // ei relevantti

        /// <summary>
        /// Poistaa kaikki ehdot täyttävät törmäyksenkäsittelijät.
        /// </summary>
        /// <param name="obj">Törmäävä olio. null jos ei väliä.</param>
        /// <param name="target">Törmäyksen kohde. null jos ei väliä.</param>
        /// <param name="tag">Törmäyksen kohteen tagi. null jos ei väliä.</param>
        /// <param name="handler">Törmäyksenkäsittelijä. null jos ei väliä.</param>
        public void RemoveCollisionHandlers( PhysicsObject obj, PhysicsObject target, object tag, ↙
    CollisionHandler<PhysicsObject, PhysicsObject> handler ) // ei relevantti
    }
}
```

H Physics2DDotNet-adapterikirjaston PhysicsGame-luokka

```
/*
 * Authors: Tero Jäntti, Tomi Karppinen, Janne Nikkanen.
 */

using System;
using Physics2DDotNet;
using Physics2DDotNet.PhysicsLogics;
using Physics2DDotNet.Joints;
using Physics2DDotNet.Solvers;
using AdvanceMath;
using System.Collections.Generic;
using Physics2DDotNet.Ignorers;
using Jypeli;

namespace AdapteriPhysics2D
{
    /// <summary>
    /// Peli, jossa on fysiikan laskenta mukana. Peliin lisätyt <code>PhysicsObject</code>-oliot
    /// käyttäytyvät fysiikan lakien mukaan.
    /// </summary>
    public class PhysicsGame : PhysicsGameBase
    {
        private GravityField gravityfield;
        private Vector gravity = Vector.Zero;

        static internal Dictionary<int, ObjectIgnorer> IgnoreGroups = null;

        /// <summary>
        /// Painovoima. Voimavektori, joka vaikuttaa kaikkiin ei-staattisiin kappaleisiin.
        /// </summary>
        public Vector Gravity
        {
            get
            {
                return gravity;
            }
            set
            {
                gravity = value;
                updatePhysicsConstants();
            }
        }

        public PhysicsLevel Level { get; set; }

        /// <summary>
        /// Alustaa uuden fysiikkapelin.
        /// </summary>
        public PhysicsGame()
            : this( 1 )
        {
        }

        /// <summary>
        /// Alustaa uuden fysiikkapelin.
        /// </summary>
        /// <param name="device">Mikä monitori käytössä, 1=ensimmäinen</param>
        public PhysicsGame( int device )
            : base( device )
        {
            Level = new PhysicsLevel(this);

            phsEngine.BroadPhase = new Physics2DDotNet.Detectors.SelectiveSweepDetector();
            //phsEngine.BroadPhase = new Physics2DDotNet.Detectors.SpatialHashDetector();

            SequentialImpulsesSolver phsSolver = new SequentialImpulsesSolver();
            phsSolver.Iterations = 12;
            phsSolver.SplitImpulse = true;
            //phsSolver.BiasFactor = 0.7;
            phsSolver.BiasFactor = 0.0;
            //phsSolver.AllowedPenetration = 0.1;
            phsSolver.AllowedPenetration = 0.01;
```

```
            phsEngine.Solver = (CollisionSolver)phsSolver;
        }

        private void updatePhysicsConstants()
        {
            if ( gravityfield != null ) gravityfield.Lifetime.IsExpired = true;

            if ( gravity != Vector.Zero )
            {
                gravityfield = new GravityField( new Vector2D( gravity.X, gravity.Y ), new Lifespan() );
                phsEngine.AddLogic( gravityfield );
            }
        }
    }
}
```

I Physics2DDotNet-adapterikirjaston PhysicsObject-luokka

```
/*
 * Authors: Tero Jäntti, Tomi Karppinen, Janne Nikkanen.
 */

using System;
using System.Diagnostics;
using AdvanceMath;
using Physics2DDotNet;
using Physics2DDotNet.Shapes;
using Physics2DDotNet.Ignorers;
using System.Collections.Generic;
using Jypeli;

#if !XBOX

#endif

namespace AdapteriPhysics2D
{
    /// <summary>
    /// Törmäyskuvion laatuun vaikuttavat parametrit.
    /// </summary>
    public struct CollisionShapeParameters // ei relevantti


    /// <summary>
    /// Kappaleen kuvion laatu törmäyksentunnistuksessa.
    /// </summary>
    [Obsolete("Use CollisionShapeParameters or the PhysicsTemplates class.")]
    public struct CollisionShapeQuality // ei relevantti

    internal class Force // ei relevantti


    /// <summary>
    /// Peliolio, joka noudattaa fysiikkamoottorin määräämiä fysiikan lakeja.
    /// Voidaan kuitenkin myös laittaa noudattamaan lakeja valikoidusti.
    /// </summary>
    public class PhysicsObject : GameObject, IPhysicsObject
    {
        internal static readonly Coefficients DefaultCoefficients = new Coefficients( 0.5, 0.4, 0.4 );
        private const double DefaultMass = 1.0;

        private double _maxAngularV = double.PositiveInfinity;
        private double _maxLinearV = double.PositiveInfinity;

        /// <summary>
        /// Olioon vaikuttavat voimat
        /// </summary>
        internal List<Force> ActiveForces = new List<Force>();

        /// <summary>
        /// Fysiikkamoottorin käyttämä tietorakenne.
        /// </summary>
        public Body Body { get; private set; }

        /// <summary>
        /// Rakenneolio, johon tämä olio kuuluu.
        /// </summary>
        public PhysicsStructure ParentStructure { get; internal set; }

        /// <summary>
        /// Olio, jolla voi välttää oliota osumasta tiettyihin muihin olioihin.
        /// </summary>
        public Ignorer CollisionIgnorer
        {
            get { return Body.CollisionIgnorer; }
            set { Body.CollisionIgnorer = value; }
        }

        int _ignoreGroup;
```

```
        /// <summary>
        /// Törmäysryhmä.
        /// Oliot jotka ovat samassa törmäysryhmässä menevät toistensa läpi.
        /// Jos ryhmä on nolla tai negatiivinen, sillä ei ole vaikutusta.
        /// </summary>
        public int CollisionIgnoreGroup
        {
            get { return _ignoreGroup; }
            set
            {
                _ignoreGroup = value;
                if ( !IsAddedToGame ) AddedToGame += AddCollisionIgnoreGroup;
                else AddCollisionIgnoreGroup();
            }
        }

        /// <summary>
        /// Olion paikka koordinaatistossa. Käsittää sekä X- että Y-koordinaatin.
        /// </summary>
        [Save]
        public override Vector Position
        {
            get
            {
                Vector2D v = Body.State.Position.Linear;
                return new Vector( v.X, v.Y );
            }
            set { Body.State.Position.Linear = new Vector2D( value.X, value.Y ); }
        }

        /// <summary>
        /// Olion koko (x on leveys, y on korkeus).
        /// </summary>
        [Save]
        public override Vector Size
        {
            get
            {
                return base.Size;
            }
            set
            {
                Body.Shape = CreatePhysicsShape( base.Shape, value );
                base.Size = value;
            }
        }

        /// <summary>
        /// Kulma, jossa olio on. Oliota voi pyörittää kulmaa vaihtamalla.
        /// </summary>
        [Save]
        public override Angle Angle
        {
            get { return Angle.FromRadians( Body.State.Position.Angular ); }
            set { Body.State.Position.Angular = value.Radians; }
        }

        /// <summary>
        /// Jos <c>false</c>, olio ei voi pyöriä.
        /// </summary>
        public bool CanRotate
        {
            get { return !double.IsPositiveInfinity( MomentOfInertia ); }
            set
            {
                if ( !value )
                {
                    MomentOfInertia = double.PositiveInfinity;
                }
                else
                {
                    UpdateBody( this.Mass );
                }
            }
```

```
        }

        /// <summary>
        /// Olion massa. Mitä suurempi massa, sitä suurempi voima tarvitaan olion liikuttamiseksi.
        /// </summary>
        /// <remarks>
        /// Massan asettaminen muuttaa myös hitausmomenttia (<c>MomentOfInertia</c>).
        /// </remarks>
        public double Mass
        {
            get { return Body.Mass.Mass; }
            set
            {
                // We should change the moment of intertia as well. If the mass is changed like, from
                // 1.0 to 100000.0, it would look funny if a heavy object would spin wildly from a small
                // touch.

                if ( !CanRotate )
                {
                    // The moment of inertia has been set to positive infinity,
                    // let's keep it that way and just set the mass.
                    Body.Mass.Mass = value;
                }
                else
                {
                    UpdateBody( value );
                }
            }
        }

        /// <summary>
        /// Olion hitausmomentti. Mitä suurempi hitausmomentti, sitä enemmän vääntöä tarvitaan
        /// olion pyörittämiseksi.
        /// </summary>
        public double MomentOfInertia
        {
            get { return Body.Mass.MomentOfInertia; }
            set { Body.Mass.MomentOfInertia = value; }
        }

        /// <summary>
        /// Lepokitka. Liikkeen alkamista vastustava voima, joka ilmenee kun olio yrittää lähteä liikkeelle
        /// toisen olion pinnalta (esim. laatikkoa yritetään työntää eteenpäin).
        /// </summary>
        public double StaticFriction
        {
            get { return Body.Coefficients.StaticFriction; }
            set { Body.Coefficients.StaticFriction = value; }
        }

        /// <summary>
        /// Liikekitka. Liikettä vastustava voima joka ilmenee kun kaksi oliota liikkuu toisiaan vasten
        /// (esim. laatikko liukuu maata pitkin). Arvot välillä 0.0 (ei kitkaa) ja 1.0 (täysi kitka).
        /// </summary>
        public double KineticFriction
        {
            get { return Body.Coefficients.DynamicFriction; }
            set { Body.Coefficients.DynamicFriction = value; }
        }

        /// <summary>
        /// Olion kimmoisuus. Arvo välillä 0.0-1.0.
        /// Arvolla 1.0 olio säilyttää kaiken vauhtinsa törmäyksessä. Mitä pienempi arvo,
        /// sitä enemmän olion vauhti hidastuu törmäyksessä.
        /// </summary>
        public double Restitution
        {
            get { return Body.Coefficients.Restitution; }
            set { Body.Coefficients.Restitution = value; }
        }

        /// <summary>
        /// Olion nopeus.
        /// </summary>
```

```
        [Save]
        public Vector Velocity
        {
            get
            {
                Vector2D v = Body.State.Velocity.Linear;
                return new Vector( v.X, v.Y );
            }
            set { Body.State.Velocity.Linear = new Vector2D(value.X, value.Y); }
        }

        /// <summary>
        /// Olion kulmanopeus.
        /// </summary>
        [Save]
        public double AngularVelocity
        {
            get { return Body.State.Velocity.Angular; }
            set { Body.State.Velocity.Angular = value; }
        }

        /// <summary>
        /// Suurin nopeus, jonka olio voi saavuttaa.
        /// </summary>
        [Save]
        public double MaxVelocity
        {
            get { return _maxAngularV; }
            set { _maxAngularV = value; IsUpdated = true; }
        }

        /// <summary>
        /// Suurin kulmanopeus, jonka olio voi saavuttaa.
        /// </summary>
        [Save]
        public double MaxAngularVelocity
        {
            get { return _maxLinearV; }
            set { _maxLinearV = value; IsUpdated = true; }
        }

        /// <summary>
        /// Olion kiihtyvyys.
        /// </summary>
        [Save]
        public Vector Acceleration
        {
            get
            {
                Vector2D v = Body.State.Acceleration.Linear;
                return new Vector( v.X, v.Y );
            }
            set { Body.State.Acceleration.Linear = new Vector2D( value.X, value.Y ); }
        }

        /// <summary>
        /// Olion kulmakiihtyvyys.
        /// </summary>
        [Save]
        public double AngularAcceleration
        {
            get { return Body.State.Acceleration.Angular; }
            set { Body.State.Acceleration.Angular = value; }
        }

        [Save]
        internal Vector ForceAccumulator
        {
            get { return new Vector( Body.State.ForceAccumulator.Linear.X, Body.State.ForceAccumulator.
    Linear.Y ); }
            set { Body.State.ForceAccumulator.Linear = new Vector2D( value.X, value.Y ); }
        }

        [Save]
```

```
        internal double AngularForceAccumulator
        {
            get { return Body.State.ForceAccumulator.Angular; }
            set { Body.State.ForceAccumulator.Angular = value; }
        }

        /// <summary>
        /// Olion muoto.
        /// </summary>
        public override Shape Shape
        {
            get { return base.Shape; }
            set
            {
                SetShape( value, GetDefaultParameters( Width, Height ) );
            }
        }

        internal void SetShape( Shape shape, CollisionShapeParameters parameters )
        {
            base.Shape = shape;
            Body.Shape = CreatePhysicsShape( shape, Size, parameters );
        }

        /// <summary>
        /// Olion hidastuminen. Hidastaa olion vauhtia, vaikka se ei
        /// osuisi mihinkään. Vähän kuin väliaineen (esim. ilman tai veden)
        /// vastus. Oletusarvo on 1.0, jolloin hidastumista ei ole. Mitä
        /// pienempi arvo, sitä enemmän kappale hidastuu.
        ///
        /// Yleensä kannattaa käyttää arvoja, jotka ovat lähellä ykköstä,
        /// esim. 0.95.
        /// </summary>
        public double LinearDamping
        {
            get { return Body.LinearDamping; }
            set { Body.LinearDamping = value; }
        }

        /// <summary>
        /// Olion pyörimisen hidastuminen.
        ///
        /// <see cref="LinearDamping"/>
        /// </summary>
        public double AngularDamping
        {
            get { return Body.AngularDamping; }
            set { Body.AngularDamping = value; }
        }

        /// <summary>
        /// Jättääkö olio räjähdyksen paineaallon huomiotta.
        /// </summary>
        public bool IgnoresExplosions { get; set; }

        /// <summary>
        /// Jättääkö olio törmäyksen huomioimatta. Jos tosi, törmäyksestä
        /// tulee tapahtuma, mutta itse törmäystä ei tapahdu.
        /// </summary>
        public bool IgnoresCollisionResponse
        {
            get { return Body.IgnoresCollisionResponse; }
            set { Body.IgnoresCollisionResponse = value; }
        }

        /// <summary>
        /// Jättääkö olio painovoiman huomioimatta.
        /// </summary>
        public bool IgnoresGravity
        {
            get { return Body.IgnoresGravity; }
            set { Body.IgnoresGravity = value; }
        }
```

```
        /// <summary>
        /// Jättääkö olio kaikki fysiikkalogiikat (ks. <c>AddPhysicsLogic</c>)
        /// huomiotta. Vaikuttaa esim. painovoimaan, mutta ei törmäyksiin.
        /// </summary>
        public bool IgnoresPhysicsLogics
        {
            get { return Body.IgnoresPhysicsLogics; }
            set { Body.IgnoresPhysicsLogics = value; }
        }

        /// <summary>
        /// Tapahtuu kun olio törmää toiseen.
        /// </summary>
        public event CollisionHandler<IPhysicsObject, IPhysicsObject> Collided;

        private void OnCollided( object sender, CollisionEventArgs args )
        {
            if ( this.IsDestroyed || args.Other == null ) return;
            var other = (PhysicsObject)args.Other.Tag;
            if ( other.IsDestroyed ) return;

            if ( Collided != null )
            {
                if ( other.ParentStructure != null ) Collided( this, other.ParentStructure );
                Collided( this, other );
            }
            Brain.OnCollision( other );
        }

        /// <summary>
        /// Tekee suorakulmaisen kappaleen, joka on staattinen (eli pysyy paikallaan).
        /// </summary>
        /// <param name="width">Kappaleen leveys</param>
        /// <param name="height">Kappaleen korkeus</param>
        /// <returns>Fysiikkaolio</returns>
        public static PhysicsObject CreateStaticObject( double width, double height ) // ei relevantti

        /// <summary>
        /// Tekee suorakulmaisen kappaleen, joka on staattinen (eli pysyy paikallaan).
        /// Kappaleen koko ja ulkonäkö ladataan parametrina annetusta kuvasta.
        /// </summary>
        /// <param name="width">Kappaleen leveys</param>
        /// <param name="height">Kappaleen korkeus</param>
        /// <returns>Fysiikkaolio</returns>
        public static PhysicsObject CreateStaticObject( Image image ) // ei relevantti

        /// <summary>
        /// Tekee vapaamuotoisen kappaleen, joka on staattinen (eli pysyy paikallaan).
        /// </summary>
        /// <param name="width">Kappaleen leveys</param>
        /// <param name="height">Kappaleen korkeus</param>
        /// <param name="shape">Kappaleen muoto</param>
        /// <returns>Fysiikkaolio</returns>
        public static PhysicsObject CreateStaticObject( double width, double height, Shape shape ) // ei ↙
    relevantti

        [Obsolete( "Use CollisionShapeParameters or the PhysicsTemplates class." )]
        public static PhysicsObject CreateStaticObject( double width, double height, Shape shape,          ↙
    CollisionShapeQuality quality ) // ei relevantti

        public static PhysicsObject CreateStaticObject( double width, double height, Shape shape,          ↙
    CollisionShapeParameters parameters ) // ei relevantti

        [Obsolete("Use function with CollisionShapeParameters")]
        public static PhysicsObject CreateStaticObject( double width, double height, Shape shape, double   ↙
    maxDistanceBetweenVertices, double gridSpacing ) // ei relevantti

        internal static PhysicsObject CreateStaticObject( double width, double height, Shape shape, IShape ↙
    physicsShape ) // ei relevantti

        /// <summary>
        /// Luo uuden fysiikkaolion.
        /// </summary>
        /// <param name="width">Leveys.</param>
```

```
        /// <param name="height">Korkeus.</param>
        public PhysicsObject( double width, double height )
            : this( width, height, Shape.Rectangle )
        {
        }

        /// <summary>
        /// Luo uuden fysiikkaolion.
        /// </summary>
        /// <param name="width">Leveys.</param>
        /// <param name="height">Korkeus.</param>
        /// <param name="shape">Muoto.</param>
        public PhysicsObject( double width, double height, Shape shape )
            : this( width, height, shape, CreatePhysicsShape( shape, new Vector( width, height ) ) )
        {
        }

        /// <summary>
        /// Luo uuden fysiikkaolion.
        /// Kappaleen koko ja ulkonäkö ladataan parametrina annetusta kuvasta.
        /// </summary>
        /// <param name="image">Kuva</param>
        public PhysicsObject( Image image )
            : this( image.Width, image.Height, Shape.Rectangle )
        {
            this.Image = image;
        }

        public PhysicsObject( double width, double height, Shape shape, CollisionShapeParameters ↙
    shapeParameters )
            : this( width, height, shape, CreatePhysicsShape( shape, new Vector( width, height ), ↙
    shapeParameters ) )
        {
        }

        [Obsolete( "Use CollisionShapeParameters or the PhysicsTemplates class." )]
        public PhysicsObject( double width, double height, Shape shape, CollisionShapeQuality quality )
            : this( width, height, shape, CreatePhysicsShape( shape, new Vector( width, height ) ) )
        {
        }

        [Obsolete( "Use constructor with CollisionShapeParameters" )]
        public PhysicsObject( double width, double height, Shape shape, double maxDistanceBetweenVertices, ↙
    double gridSpacing )
            : this( width, height, shape, new CollisionShapeParameters( maxDistanceBetweenVertices, ↙
    gridSpacing ) )
        {
        }

        /// <summary>
        /// Luo fysiikkaolion, jonka muotona on säde.
        /// </summary>
        /// <param name="raySegment">Säde.</param>
        public PhysicsObject( Jypeli.RaySegment raySegment )
            : this( 1, 1, raySegment )
        {
        }

        /// <summary>
        /// Initializes the object with the given physics shape. The size of
        /// the physicsShape must be the one given.
        /// </summary>
        internal PhysicsObject( double width, double height, Shape shape, IShape physicsShape )
            : base( width, height, shape )
        {
            Coefficients c = new Coefficients( DefaultCoefficients.Restitution, DefaultCoefficients. ↙
    StaticFriction, DefaultCoefficients.DynamicFriction );
            Body = new Body( new PhysicsState( ALVector2D.Zero ), physicsShape, DefaultMass, c, new Lifespan ↙
    () );
            Body.Tag = this;
            Body.Collided += this.OnCollided;
        }

        private void AddCollisionIgnoreGroup()
```

```
        {
            if ( _ignoreGroup == 0 )
            {
                Body.CollisionIgnorer = null;
            }
            else
            {
                if ( PhysicsGame.IgnoreGroups == null ) PhysicsGame.IgnoreGroups = new Dictionary<int, ↙
    ObjectIgnorer>();
                if ( !PhysicsGame.IgnoreGroups.ContainsKey( _ignoreGroup ) ) PhysicsGame.IgnoreGroups.Add( ↙
    _ignoreGroup, new ObjectIgnorer() );
                Body.CollisionIgnorer = PhysicsGame.IgnoreGroups[_ignoreGroup];
            }
        }

        private static CollisionShapeParameters GetDefaultParameters( double width, double height )
        {
            CollisionShapeParameters p;
            p.MaxVertexDistance = Math.Min( width, height ) / 3;
            p.DistanceGridSpacing = Math.Min( width, height ) / 2;
            return p;
        }

        /// <summary>
        /// Tekee oliosta staattisen. Staattinen olio ei liiku muiden olioiden törmäyksistä,
        /// vaan ainoastaan muuttamalla suoraan sen paikkaa tai nopeutta.
        /// </summary>
        public void MakeStatic()
        {
            Mass = double.PositiveInfinity;
            CanRotate = false;
            IgnoresGravity = true;
        }

        /// <summary>
        /// This updates the mass and momentOfInertia based on the new mass.
        /// </summary>
        private void UpdateBody( double mass )
        {
            Body.Mass = Body.GetMassInfo( mass, Body.Shape );
        }

        internal static IShape CreatePhysicsShape( Shape shape, Vector size )
        {
            return CreatePhysicsShape( shape, size, GetDefaultParameters( size.X, size.Y ) );
        }

        /// <summary>
        /// Creates a shape to be used in the Physics Body. A physics shape is scaled to the
        /// size of the object. In addition, it has more vertices and some additional info
        /// that is used in collision detection.
        /// </summary>
        internal static IShape CreatePhysicsShape( Shape shape, Vector size, CollisionShapeParameters ↙
    parameters )
        {
            if ( shape is Jypeli.RaySegment )
            {
                Jypeli.RaySegment raySegment = (Jypeli.RaySegment)shape;
                Physics2DDotNet.Shapes.RaySegment singleSegment = new Physics2DDotNet.Shapes.RaySegment(
                    new Vector2D(raySegment.Origin.X, raySegment.Origin.Y),
                    new Vector2D(raySegment.Direction.X, raySegment.Direction.Y),
                    raySegment.Length );
                return new RaySegmentsShape( singleSegment );
            }
            else if ( shape is Ellipse )
            {
                Debug.Assert( shape.IsUnitSize );

                double smaller = Math.Min( size.X, size.Y );
                double bigger = Math.Max( size.X, size.Y );
                // Average between width and height.
                double r = smaller / 2 + ( bigger - smaller ) / 2;
                int vertexCount = (int)Math.Ceiling( ( 2 * Math.PI * r ) / parameters.MaxVertexDistance );
```

```
            if ( Math.Abs(size.X - size.Y) <= double.Epsilon )
            {
                // We get more accurate results by using the circleshape.
                // in addition, the circleshape does not need a DistanceGrid
                // object (which is slow to initialize) because calculations
                // for a circleshape are much simpler.
                return new CircleShape( r, vertexCount );
            }
            else
            {
                Vector2D[] vertexes = new Vector2D[vertexCount];

                double a = 0.5 * size.X;
                double b = 0.5 * size.Y;

                for ( int i = 0; i < vertexCount; i++ )
                {
                    double t = ( i * 2 * Math.PI ) / vertexCount;
                    double x = a * Math.Cos( t );
                    double y = b * Math.Sin( t );
                    vertexes[i] = new Vector2D( x, y );
                }

                return new PolygonShape( vertexes, parameters.DistanceGridSpacing );
            }
        }
        else
        {
            Vector2D[] originalVertexes = new Vector2D[shape.Cache.OutlineVertices.Length];
            for ( int i = 0; i < shape.Cache.OutlineVertices.Length; i++ )
            {
                Vector v = shape.Cache.OutlineVertices[i];
                if ( shape.IsUnitSize )
                {
                    v.X *= size.X;
                    v.Y *= size.Y;
                }
                originalVertexes[i] = new Vector2D(v.X, v.Y);
            }

            Vector2D[] polyVertexes = VertexHelper.Subdivide(originalVertexes, parameters.      ↙
MaxVertexDistance);

            return new PolygonShape(polyVertexes, parameters.DistanceGridSpacing);
        }
    }

    /// <summary>
    /// Työntää oliota.
    /// </summary>
    /// <param name="force">Voima, jolla oliota työnnetään.</param>
    public virtual void Push( Vector force )
    {
        Body.ApplyForce(new Vector2D(force.X, force.Y));
    }

    /// <summary>
    /// Työntää oliota tietyn ajan tietyllä voimalla.
    /// </summary>
    /// <param name="force">Voima, jolla oliota työnnetään sekunnissa.</param>
    /// <param name="time">Aika, kuinka kauan voimaa pidetään yllä.</param>
    public virtual void Push( Vector force, TimeSpan time )
    {
        IsUpdated = true;
        Body.ApplyForce( new Vector2D( force.X, force.Y ) );
        ActiveForces.Add( new Force( force, time ) );
    }

    /// <summary>
    /// Kohdistaa kappaleeseen impulssin. Tällä kappaleen saa nopeasti liikkeeseen.
    /// </summary>
    public virtual void Hit( Vector impulse )
    {
        Body.ApplyImpulse(new Vector2D(impulse.X, impulse.Y));
```

```
        }

        /// <summary>
        /// Kohdistaa kappaleeseen vääntövoiman. Voiman suunta riippuu merkistä.
        /// </summary>
        /// <param name="torque">Vääntövoima.</param>
        public virtual void ApplyTorque( double torque )
        {
            Body.ApplyTorque(torque);
        }

        /// <summary>
        /// Pysäyttää olion.
        /// </summary>
        public virtual void Stop()
        {
            Body.ClearForces();
            Body.State.Acceleration = ALVector2D.Zero;
            Body.State.Velocity = ALVector2D.Zero;
            Body.State.ForceAccumulator = ALVector2D.Zero;
            StopMoveTo();
        }

        /// <summary>
        /// Pysäyttää MoveTo-aliohjelmalla aloitetun liikkeen.
        /// </summary>
        public void StopMoveTo()
        {
            if ( moveTimer != null )
            {
                moveTimer.Stop();
                moveTarget = null;
            }
        }

        /// <summary>
        /// Pysäyttää olion liikkeen vaakasuunnassa.
        /// </summary>
        public void StopHorizontal()
        {
            ALVector2D oldAcc = Body.State.Acceleration;
            Body.State.Acceleration = new ALVector2D( oldAcc.Angular, 0, oldAcc.Y );

            ALVector2D oldVel = Body.State.Velocity;
            Body.State.Velocity = new ALVector2D( oldVel.Angular, 0, oldVel.Y );

            ALVector2D oldForce = Body.State.ForceAccumulator;
            Body.State.ForceAccumulator = new ALVector2D( oldForce.Angular, 0, oldForce.Y );
        }

        /// <summary>
        /// Pysäyttää olion liikkeen pystysuunnassa.
        /// </summary>
        public void StopVertical()
        {
            ALVector2D oldAcc = Body.State.Acceleration;
            Body.State.Acceleration = new ALVector2D( oldAcc.Angular, oldAcc.X, 0 );

            ALVector2D oldVel = Body.State.Velocity;
            Body.State.Velocity = new ALVector2D( oldVel.Angular, oldVel.X, 0 );

            ALVector2D oldForce = Body.State.ForceAccumulator;
            Body.State.ForceAccumulator = new ALVector2D( oldForce.Angular, oldForce.X, 0 );
        }

        public override void Destroy()
        {
            Body.Lifetime.IsExpired = true;
            base.Destroy();
        }

        public override void Update( Time time )
        {
            if ( Velocity.Magnitude > MaxVelocity )
```

```
                Velocity = Vector.FromLengthAndAngle( MaxVelocity, Velocity.Angle );
            if ( AngularVelocity > MaxAngularVelocity )
                AngularVelocity = MaxVelocity;

            for ( int i = ActiveForces.Count - 1; i >= 0; i-- )
            {
                if ( ActiveForces[i].IsDestroyed() )
                {
                    ActiveForces.RemoveAt( i );
                    continue;
                }

                // Apply the force
                Push( ActiveForces[i].Value * time.SinceLastUpdate.TotalSeconds );
            }

            base.Update( time );
        }

        /// <summary>
        /// Siirtää oliota.
        /// </summary>
        /// <param name="movement">Vektori, joka määrittää kuinka paljon siirretään.</param>
        public override void Move( Vector movement )
        {
            Vector dv = movement - this.Velocity;
            Hit( Mass * dv );
        }

        protected override void MoveToTarget()
        {
            if ( !moveTarget.HasValue )
            {
                Stop();
                moveTimer.Stop();
                return;
            }

            Vector d = moveTarget.Value - Position;
            double vt = moveSpeed * moveTimer.Interval;

            if ( d.Magnitude < vt )
            {
                Vector targetLoc = moveTarget.Value;
                Stop();
                moveTimer.Stop();
                moveTarget = null;
                OnArrived( targetLoc );
            }
            else
            {
                Vector dv = Vector.FromLengthAndAngle( moveSpeed, d.Angle ) - this.Velocity;
                Hit( Mass * dv );
            }
        }
    }
}
```