Teemu Hörkkö

Komponenttipohjainen objektinhallinta pelinteossa

Opinnäytetyö
Kajaanin ammattikorkeakoulu
Tradenomi
Tietojenkäsittelyn koulutusohjelma / Peliala
Syksy 2014

KAJAANIN AMMATTIKORKEAKOULU
UNIVERSITY OF APPLIED SCIENCES

OPINNÄYTETYÖ
TIIVISTELMÄ

Koulutusala Tradenomi	Koulutusohjelma Tietojenkäsittelyn tradenomi
Tekijä(t) Teemu Hörkkö	
Työn nimi Komponenttipohjainen objektinhallinta pelinteossa	
Vaihtoehtoiset ammattiopinnot Peliohjelmointi	Toimeksiantaja -
Aika Syksy 2014	Sivumäärä ja liitteet 23

Opinnäytetyön tavoitteena oli tutkia ja esitellä komponenttipohjaiseen arkkitehtuuriin perustuvan objektinhallinnan perusteita erityisesti pelinkehityksen näkökulmasta. Ensin tarkoitus oli kirjoittaa pelimoottorin kehityksestä, mutta opinnäytetyön aihe rajautui vain komponenttipohjaiseen objektinhallintaan. Opinnäytetyön aihealue on silti ajankohtainen ja keskeinen osa nykyaikaisia pelimoottoreita.

Opinnäytetyön alussa esitellään perinteisen kehitystavan hyviä ja huonoja puolia ja vaihtoehtoinen lähestymistapa hyödyntämällä modulaarisempia komponentteja. Komponentteihin perustuva pelinkehitys on nopeampaa kuin perinteinen kankeampi luokkaperintään pohjautuva ohjelmointitapa. Modulaarisempi ohjelmistoarkkitehtuuri soveltuu paremmin pelintekoon, sillä peleissä tarvitaan useita eri objektityyppejä ja niiden rakentelu ja testaus on nopeampaa osista rakennettuina.

Lisäksi opinnäytteen aikana toteutettiin myös yksinkertainen loogisiin järjestelmiin perustuva komponenttipohjainen objektinhallinnan toteutus C++-ohjelmointikielellä. Opinnäytetyössä käydään alustavasti läpi luokkakaavioin ja koodiesimerkein sen toteutusta ja käyttämistä.

Kieli	Suomi
Asiasanat	objektinhallinta, entiteetti, komponentti, järjestelmä
Säilytyspaikka	☒ Verkkokirjasto Theseus ☐ Kajaanin ammattikorkeakoulun kirjasto

KAJAANIN AMMATTIKORKEAKOULU
UNIVERSITY OF APPLIED SCIENCES

THESIS
ABSTRACT

School Business	Degree Programme Bachelor of Business Administration
Author(s) Teemu Hörkkö	
Title Component-Based Object Management in Game Development	
Optional Professional Studies Game Programming	Commissioned by -
Date Autumn 2014	Total Number of Pages and Appendices 23

The goal of this thesis was to research and present the basics behind a component-based object management from the perspective of game development. My intention at first was to write about game engine development but the subject was refocused to only object systems instead. The thesis subject, however, is still relevant and central part of modern game engine development practices.

Thesis reviews the pros and the cons of the traditional object management and offers an alternative option with the help of modular components. Game development based on component systems is faster compared to traditional rigid class inheritance. The modular software architecture lends itself better to game development where several types of objects are required and it is more efficient to build them from parts.

In addition the thesis also presents a simple implementation of component-based object management with logic systems made during the thesis process using C++ programming language. The thesis tentatively goes through the different parts of the implementation and its use with class diagrams and a few code examples.

Language of Thesis	Finnish
Keywords	object management, entity, component, system
Deposited at	☒ Electronic library Theseus ☐ Library of Kajaani University of Applied Sciences

SISÄLLYS

KÄYTETYT TERMIT JA LYHENTEET

Abstrakti metodi	Osa polymorfismia, abstrakti metodi on esitelty luokan jäsenfunktio, mutta sillä ei ole toteutusta vaan perivän luokan on tarkoitus määrittää se.
Aksessorifunktio	Eng. accessor function. Luokan julkinen funktio jonka ainut käyttötarkoitus on olla ohjelmoijalle turvallinen tapa hakea tai muokata luokan yksityisiä ja suojattuja parametreja.
Constructor ja Destructor	Luokan alustus- ja tuhoamismetodit. Molempia kutsutaan automaattisesti silloin kun se on ajankohtaista.
JSON	Lyhenne sanoista JavaScript Object Notation, JSON avoimen standardin tiedostomuoto tiedonvälitykseen.
Kirjasto	Tietotekniikassa kirjastolla tarkoitetaan yleensä ohjelma- tai luokkakirjastoa joka on kokoelma luokkia, aliluokkia ja ohjelmia helpottamaan tietokoneohjelmiston modulaarista kehittämistä.
Olio-ohjelmointi	Olio-ohjelmointi tai englanniksi Object Oriented Programming on ohjelmoinnin lähestymistapa jossa ohjelmointiongelmia ratkotaan kuvaamalla asioita olioina joilla on toisiinsa liittyvää tietoa. Olioluokat voivat periä toisiltaan yleisiä ominaisuuksia.
Polymorfismi	Polymorfismissa luodaan yksi tunnettu käyttöliittymä jota voidaan käyttää useampien perittyjen luokkien hallintaa varten.
Staattinen metodi tai muuttuja	Luokan jäsenfunktioita ja -muuttujia voidaan määritellä staattisiksi eikä niiden kutsuminen tai käsittely vaadi luokasta instanssia.
Template	C++:ssa template-metodit ja -luokat mahdollistavat geneeristen tyyppien käyttämisen tunnettujen sijasta. Näin metodit ja luokat voivat toimia useiden datatyyppien kanssa ilman, että se täytyy erikseen ohjelmoida.
XML	Lyhenne sanoista Extensible Markup Language, XML on rakenteellinen kuvauskieli jolla voidaan kuvata tietoa ja jäsentää laajoja tietomassoja selkeämmin.

1 JOHDANTO

Idea jakaa eri objektien piirteet omiin kokonaisuuksiinsa lähti liikkeelle joskus vuosituhannen vaihteen tietämissä, mutta sille ei ole nimitetty selvää keksijää eikä se ole varsinaisesti edes kovin uusi idea (Pie21 2011). Komponenttipohjainen arkkitehtuuri soveltuu myös muuhunkin kehitystyöhön kuin vain pelinkehittämiseen, mutta perinteinen olio-ohjelmointitapa soveltuu huomattavasti huonommin peliohjelmointiin. Kehittämistyön helpottaminen ja nopeuttaminen on iso juttu, sillä peliprojektit alati paisuvat isommaksi ja niiden kehitystyö maksaa huimasti.

Komponenttipohjainen objektinhallinta pelinkehityksessä on mielenkiintoinen tapa toteuttaa pelimaailman objektit ja itse kiinnostuin aiheeseen hieman vajaan johdannon jälkeen oppitunnilla. Aloin kehittämään omia järjestelmiä sillä hetkellä kuluneisiin projekteihin, mutta vasta myöhemmin olen ymmärtänyt idean kokonaisuudessaan.

Työn tavoitteena on kuvata komponenttipohjaisen arkkitehtuurin ideaa pelinkehityksessä. Työ painottuu teorian läpikäymiseen yhdestä toteutustavasta ja yksinkertaisen toteutuksen luomiseen esitetyn teorian pohjalta.

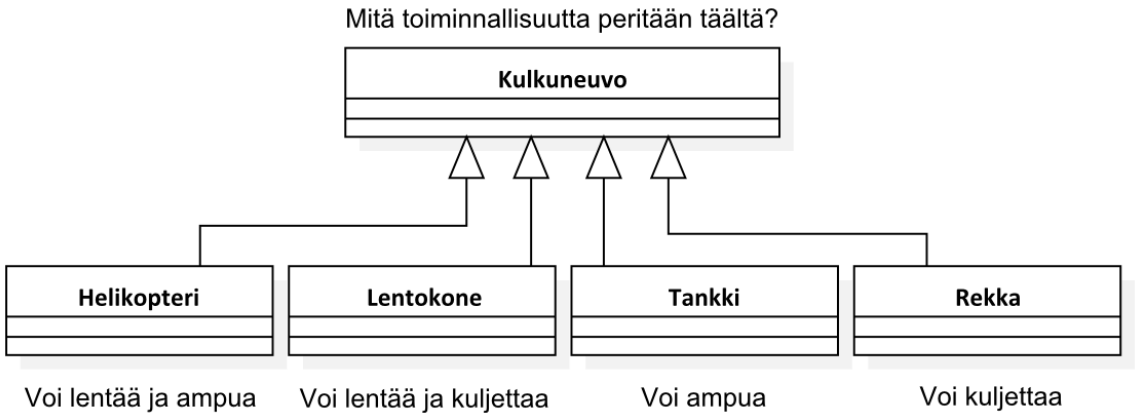
2 PELIEN OHJELMISTOARKKITEHTUURI

2.1 Tyypillinen perintään pohjautuva arkkitehtuuri

Peliohjelmoinnissa on perinteisesti kuvattu peliobjekteja olioina olio-ohjelmointiperiaatteen hierarkkiseen perintään pohjautuvalla mallilla. Ohjelmoija voi kuvata eri objektit omalla luokallansa, ja objektien välisiä samankaltaisuuksia voi jakaa ylempiin kantaluokkiin, joista peritään ominaisuuksia eteenpäin. Esimerkiksi tankki ja rekka ovat molemmat kulkuneuvoja, joten on järkevää jakaa yhteinen toiminnallisuus abstraktimpaan Kulkuneuvo-luokkaan (ks. kuvio 1 seuraavalla sivulla). Yleensä geneerisimmät objektit ovat korkeammalla hierarkiassa yläpäässä ja enemmän spesialisoituneet matalammalla. Eri objektien ominaisuudet voidaan olio-ohjelmointimallissa kuvata vain olion funktioina. Monet peliohjelmointiin käytettävät kielet tukevat olio-ohjelmointia, ja erityisesti sitä käytettiin 90-luvulla. (Pie21 2011; Porter, N. 2012.)

Tällä tavalla eri asioiden jako omiin olioihinsa on ohjelmoijalle luonteva ja suhteellisen suoraviivainen tapa hallita erilaisia peliobjekteja. Erilaiset pelimaailman objektit on jaoteltu itseään kuvaaviin olioihin. Ongelma perinnässä ilmenee vasta kun tulee aika lisätä useita erilaisia objektityyppejä peliin. Usein lähdekoodia saattaa joutua muuttaman huomattavasti jos haluaa välttää turvautumista huonoihin toteutuskeinoihin. Saman toiminnallisuuden kopiointi usealle objekteille tekee koodista vaikeammin ylläpidettävää ja myöhemmin uudelleenkäytettävää. Ohjelmoija voi myös siirtää ominaisuuksia abstraktimmille kantaluokille, mutta muut objektityypit saattavat siten periä niille tarpeetonta toiminnallisuutta. Lentokoneetkin ovat kulkuneuvoja ja ne voivat lentää, toisin kuin tankit ja rekat, ja jos lentämisen toiminnallisuus laitetaan Kulkuneuvo-luokkaan, sen saavat myös ne. Toiminnallisuuden luonti vain lentokoneille ei myöskään ole paras vaihtoehto, sillä myös helikopterit voivat lentää. (Pie21 2011; Porter, N. 2012; Crombecq, K. 2011.)

Kuvio 1 näyttää yksinkertaistetun esimerkin hierarkiasta, jossa on edellä kuvailtu tapaus. Ohjelmoijalla on ongelmana, mitä eri kulkuneuvojen ominaisuuksista jaetaan abstraktimmalle kantaluokalle, mutta kuitenkin niin, ettei niitä peritä tarpeettomasti objekteille, jotka eivät sitä tarvitse. (Boreal Games 2013; Crombecq, K. 2011.)



Kuvio 1. Yksinkertainen esimerkki kulkuneuvojen hierarkiasta

Yksinkertaisissa peleissä vain muutamilla säännöillä ja objekteilla perintä saattaa toimia tiettyyn pisteeseen saakka. On yleensä suhteellisen nopeaa ohjelmoida olioilla ja monet aloittelijoille tarkoitetut oppaat tapaavat opettaa aloittavia peliohjelmoijia tähän. Perinnän riittämätön skaalautuvuus isompiin projekteihin ei kuitenkaan tee siitä hyvää vaihtoehtoa isompiin projekteihin. (Porter, N. 2012.)

Hierarkkinen malli ei ole modulaarinen. Oliot määrittävät omat ominaisuutensa ja ovat siten tiukasti sidottuja useisiin pelimoottorin alijärjestelmiin. Malli ei myöskään mahdollista objektien luontia datasta, ja esimerkiksi C++:aa käyttäessä se tarkoittaa, että kaikki muutokset mihin tahansa osaan hierarkiasta tarkoittaa koko hierarkian uudelleenkääntämistä, joka kasvattaa monesti jo pitkiä kääntämisaikoja. (Porter, N. 2012.)

Hyvän peliobjektijärjestelmän tulisi olla helposti skaalautuva ja eri tyyppien lisääminen yhtä helppoa huolimatta siitä, kuinka monta erilaista tyyppiä pelissä jo on. Ideaalisesti eri ominaisuuksien uudelleenkäyttö eri objekteille tai jopa peleille modulaarisesti olisi helppoa. On myös tärkeää, että eri objektityyppien poisto ei aiheuta ongelmia. Siksi tyypillinen hierarkkinen malli ei ole riittävä tähän käyttötarkoitukseen. (Porter, N. 2012.)

On siis selvää, että tähän täytyy löytää erilainen ratkaisu, joka on myös riittävä tulevaisuudessa. Sen sijaan, että objektit luodaan perimällä kantaluokkia, voidaan niitä kuvailla modulaarisemmilla komponenteilla. (Porter, N. 2012.)

2.2 Komponenttipohjainen arkkitehtuuri

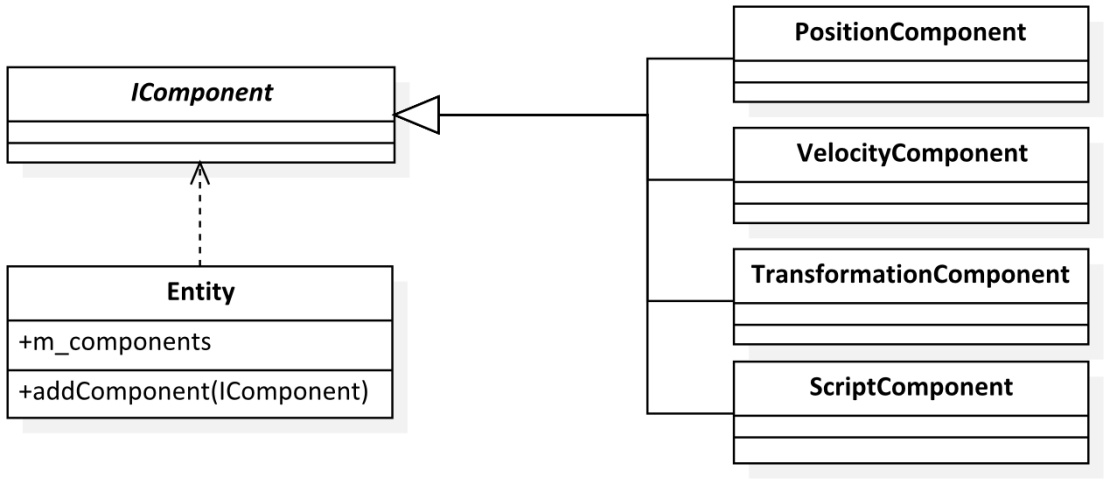
Modernissa ohjelmistosuunnittelussa on tärkeää lähdekoodin modulaarisuus eli kokonaisuuden eri ominaisuuksien jakaminen itsenäisiin komponentteihin, joilla voidaan rakentaa kokonaisuus takaisin yhteen. Modulaarisen ratkaisun joustavuus ja skaalautuvuuden perusta on, että kun jotakin osaa täytyy muuttaa, se ei vaikuta kokonaisuuden toiminnallisuuteen lainkaan. Niin kauan kuin komponenttien välinen kommunikaatio pysyy samana, ne ovat täysin tietämättömiä muutoksesta. (Pie21 2011.)

Komponenttipohjaisessa arkkitehtuurissa perintähierarkia tasoitetaan yhteen objektityyppiin, joka sisältää listan komponenteistaan. Ominaisuudet jaetaan omiin komponentteihinsa, jotka määrittävät objektin eli entiteetin piirteitä. Varsinainen looginen toiminnallisuus on ohjelmoitu vasta järjestelmiin, tai jossain implementaatiotavoissa komponentteihin. (Porter, N. 2012; Boreal Games 2013; Pie21 2011.)

Entiteetti on asia tai olio pelimaailmassa. Käytännössä se käyttäytyy täysin samalla tavalla kuin oliot olio-ohjelmointiperiaatteessa. Jokainen asia maailmassa on oma entiteettinsä: jos pelissä on sata tankkia, on olemassa myös sata entiteettiä. Lähdekoodin puolelta entiteetit voivat olla oma luokkansa, mutta niillä ei tavallisesti ole dataa tai omia ominaisuuksia ja ne voidaan korvata vain tunnistenumerolla. Semmoisenaan entiteetit ovatkin täysin hyödyttömiä, sillä vasta komponentit määrittelevät entiteetin ominaisuudet. (Pie21 2011; Martin, A. 2007.)

Objekteilla on useita eri piirteitä, jotka tekevät siitä sen mitä se on ja määrittelevät sen käyttäytymisen maailmassa. Esimerkiksi tankki on tehty metallista, ihminen voi ajaa sitä, sillä voi ampua, sen voi myydä tai ostaa ja se on ihmisten tekemä. Komponentit kuvaavat näitä piirteitä. Yksi komponentti on modulaarinen ja geneerinen palikka, joka entiteettiin liitettynä määrittää sen ominaisuuksia. Yksi komponentti ei välttämättä ole vain yksi piirre sillä entiteetin komponentti vain ilmaisee, että entiteetillä on kyseinen piirre ja järjestelmät toteuttavat varsinaisen loogisen toiminnallisuuden. (Martin, A. 2007.)

Kuviossa 2 näytetään muutama eri mahdollinen komponenttityyppi ja suhde entiteettiin.



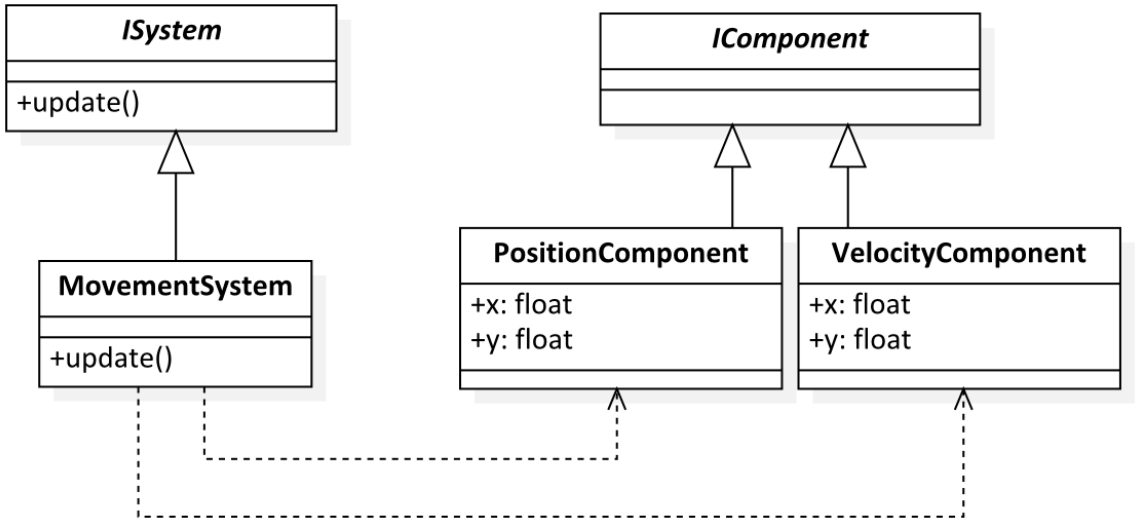
Kuvio 2. Eri komponentteja ja entiteetti

Komponenttien kanssa hierarkia muuttuu lähes kokonaan litteäksi. Jokainen komponenteista kuvaa entiteetin piirrettä tai ominaisuutta, ja yhdessä niistä muodostuu kokonaisuus. Eri komponentit sisältävät sen piirteen datan, eli esimerkiksi PositionComponent sisältää objektin sijainnin maailmassa. Yhdessä PositionComponent ja VelocityComponent mahdollistavat objektin liikkumisen pelimaailmassa. Komponenteilla itsellään ei ole mitään ulkoisesti vaikuttavia metodeita. (Boreal Games 2013; Porter, N. 2012.)

Järjestelmät implementoivat varsinaisen toiminnallisuuden komponenttien pohjalta. Komponenttien ilmaisemat piirteet aktivoivat eri järjestelmät ajamaan niiden toteuttamaa käyttäytymistä. Esimerkiksi sijainti- ja nopeuskomponentit yhdessä järjestelmän kautta muodostuvat liikejärjestelmäksi tai niin kutsuttu MovementSystem päivittää entiteetin sijaintia maailmassa sen nopeuskomponentin datan mukaan. (Pie21 2011.)

Jokaista järjestelmää ajetaan taustalla (vähän kuin niillä olisi omat säikeensä), ja ne päivittävät globaaleja toimintoja jokaisella entiteetillä, jolla on samat komponentit joita järjestelmä implementoi. Järjestelmien ja saatavilla olevien komponenttien välillä on yksi yhteen -suhde. Järjestelmät toteuttavat komponenttien metodit, toisin kuin perinteisessä olio-ohjelmoinnissa, jossa jokainen komponentti omistaa metodinsa. (Martin, A. 2007.)

Kuvio 3 esittää liikejärjestelmän ja sen tukemat komponentit.

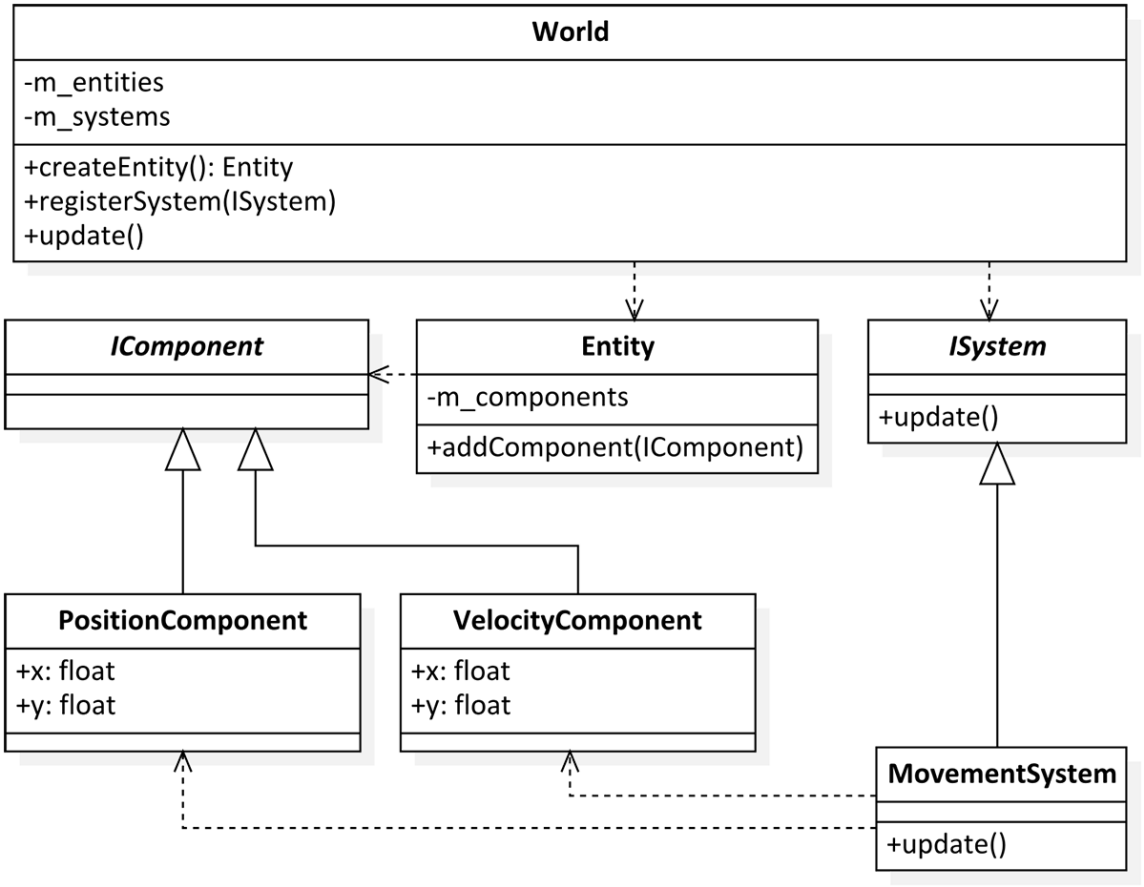


Kuvio 3. Komponentteja ja niihin liittyvä järjestelmä

Komponentit toimivat vain datan säilyttäjinä ja entiteetin piirteiden tai ominaisuuksien kuvaajina järjestelmille. Järjestelmät hyödyntävät ja päivittävät hallitsemiaan komponentteja. Jos entiteetillä ei olisi kaikkia komponentteja, joita järjestelmä vaatii, ei kyseinen järjestelmä päivitä niitä komponentteja. (Boreal Games 2013; Gamadu 2012.)

Kaikki edellä mainitut osat yhdistyvät yleiseen globaaliin pelimaailman hallintaan tarkoitetussa managerissa. Sen avulla luodaan ja hallitaan entiteettejä ja manageri hoitaa myös eri järjestelmien päivityksen pelin pääsilmukassa. Pelimaailman manageri voi myös ylläpitää pelimaailman fysiikoita. (Gamadu 2012.)

Kuvio 4 esittää yksinkertaistetun luokkakaavion mahdollisesta maailmamanagerin toteuttamistavasta. Itse maailmamanageri hallinnoi kaikkia entiteettejä ja järjestelmiä ja sen kautta voidaan myös luoda (ja poistaa) uusia entiteettejä. (Gamadu 2012; Pie21 2011.)



Kuvio 4. Maailmamanagerin suhteet muihin luokkiin koko hierarkiassa

2.3 Dynaaminen kompositio

Peliobjektien luonti erillisistä komponenteista tekee mahdolliseksi helpottaa ja nopeuttaa objektien toteuttamista käyttäen dynaamista kompositiota. Sen sijaan, että entiteetin ominaisuudet olisivat lähdekoodiin kovakoodattuja, ne tallennetaankin tiedostoon. Tällainen mallinetiedosto voidaan ladata lennosta ja luoda sen kuvaama entiteetti määritellyillä komponenteilla ja oletusarvoilla. Dynaaminen kompositio tekee myös eri ideoiden testaamisesta helpompaa, sillä koko pelin lähdekoodia ei tarvitse kääntää joka kerta uudestaan ja peliohjelmoijien ei ole pakko olla aina mukana lisäämässä, muuttamassa tai poistamassa ominaisuuksia. Tämä on erityisesti totta, jos dynaaminen kompositio yhdistetään jonkin skriptikielen kanssa, esimerkiksi Lua tai Squirrel. Eri komponenttien tai järjestelmien toiminnallisuus voidaan ohjelmoida skriptikielellä, jota tulkataan samaan aikaan pelin ollessa käynnissä. (Crombecq, K. 2011; Porter, N. 2012.)

Entiteetin mallinetiedostoformaatiksi sopii erinomaisesti esimerkiksi XML tai JSON. Malline listaa entiteetin piirteet eli komponentit ja komponenttien oletusarvot. Tiedostoon voidaan myös listata muuta entiteettiin liittyvää tietoa, kuten sen vaatimia resursseja, muun muassa tekstuureita tai ääninäytteitä. (Crombecq, K. 2011.)

Malline esimerkki listauksessa 1 kuvaa vihollistankkia ja sille on lisätty useita eri komponentteja ja niiden oletusarvot. Entiteetin lataava pelimoottorin funktio sitten luo tästä mallineesta uuden pelimaailman objektin. (Crombecq, K. 2011.)

```
<entity ref="EnemyTank">
    <components>
        <position x="0" y="0" />
        <velocity x="0" y="0" />
        <transform>
            <scale x="1.0" y="1.0" />
            <rotation angle="0" />
        </transform>
        <collision type="rectangle" x="40" y="50" />
        <renderer>
            <texture>EnemyTank.png</texture>
            <origin anchor="center" x="0" y="0" />
        </renderer>
        <script file="EnemyTank.lua" />
    </components>
</entity>
```

Listaus 1. XML-pohjainen entiteettimalline voi näyttää tältä.

2.4 Eri komponentti- ja järjestelmätyyppejä ja niiden käyttö

Pelejä varten on useita eri komponentti ja järjestelmätyyppejä, joista monet voivat olla vain yhteen peliin tai tarkoitukseen soveltuvia. On silti olemassa useampia geneerisempiä ja uudelleenkäytettäviä komponentti- ja järjestelmätyyppejä, joita todennäköisesti löytyy joka pelistä. (Boreal Games 2013; Pie21 2011.)

Komponentit toimivat entiteettien merkkaamiseen ja niiden ominaisuuksien määrittelyyn, jotta järjestelmät osaavat toteuttaa ominaisuuksien käytöksen. Parametrittomat komponentit toimivat entiteetin merkkaamiseen yhtä ainoaa asiaa varten. Esimerkiksi pelin kamera tai pelaajan hahmo voidaan merkitä vain yksinkertaisella komponentilla ilman erillistä dataa. Tällä tavoin kameran tai pelaajan hahmon järjestelmät osaavat käsitellä vain näitä entiteettejä. Tau-

lukossa 1 seuraavalla sivulla on listattu esimerkkinä muutamia mahdollisia komponenttityyppejä, niihin liittyvät parametrit ja käyttötarkoitus. (Boreal Games 2013.)

Taulukko 1. Esimerkkejä komponenteista ja käyttötarkoituksista. (Boreal Games 2013.)

Komponentti	**Parametrit**	**Käyttötarkoitus**
Position	X, Y ja Z (jos 3d)	Entiteetin koordinaatit
Velocity	X, Y ja Z (jos 3d)	Entiteetin nopeus
Physics	Body (fysiikkamoottori)	Fysiikkamoottorin body
Sprite	Tekstuuri/animaatiot	Merkkaa entiteetin piirrettäväksi ja sisältää tekstuuri/animaatiodatan
Health/Energy	Arvo	Yksinkertaisesti säilyttää yhden arvon
Character	Nimi, taso	Säilyttää tietoa pelihahmoista
Player	Ei parametreja	Ainoastaan merkkaa entiteetin pelaajan hahmoksi
Input	Ei parametreja	Merkkaa syötteitä vastaanottavaksi

Pelkästään jo näillä komponenteilla voidaan luoda useampia erilaisia entiteettejä. Esimerkiksi liikkumaton kivi käyttäisi vain positio- ja sprite-komponentteja. Pelaaja tai vihollinen käyttäisi huomattavasti useampia. (Boreal Games 2013.)

Taulukko 2 esittelee järjestelmiä, jotka toteuttavat taulukossa 1 esiteltyjen komponenttien toiminnallisuutta.

Taulukko 2. Järjestelmiä ja niiden komponentit (johdettu Boreal Games 2013.)

Järjestelmä	**Komponentit**	**Käyttötarkoitus**
Movement	Position ja Velocity	Päivittää sijaintia nopeuden mukaan
Physics	Position, Velocity ja Physics	Päivittää fysiikoita
Renderer	Position ja Sprite	Entiteetin piirto ruudulle
AI	Position, Velocity, Health ja Character	Tekoäly
Player	Input, Velocity ja Player	Käsittelee pelaajan syötteet

2.5 Yleisiä toteutustapoja

Komponenttipohjaisen arkkitehtuurin toteuttamiseen on useampia eri tapoja, mutta kaksi yleisintä toteutustapaa eroavat loogisen toiminnallisuuden sijainnissa. Aiemmin tapana oli loogisen toiminnallisuuden toteuttaminen komponenteissa ilman järjestelmiä ja komponentit hoitivat viestityksen keskenään. Toinen toteutustapa on jo aiemmin esitelty järjestelmiin pohjautuva arkkitehtuuri. (Crombecq, K. 2011; Boreal Games 2013.)

Loogisen toiminnallisuuden toteuttaminen komponentteihin hyödyntää silti olio-ohjelmoinnin etuja, mutta sillä vältetään tarpeeton syvä luokkahierarkia. Komponentti itsessään kapseloi ominaisuudet yhteen pakettiin. Tällä toteutustavalla komponentit eivät silti ole täysin itsenäisiä, sillä ne usein tarvitsevat samoja resursseja, joita toinen komponentti omistaa. Esimerkiksi tekoälykomponentti todennäköisesti haluaa liikuttaa objektia, jolloin se tarvitsee positiokomponentin. Komponentit voivat vaihtaa tietoa keskenään viestimällä suorasti tai lähettämällä viestejä tai tapahtumia. (Crombecq, K. 2011.)

Suora viestiminen perustuu komponenttien täyteen riippuvuuteen toisistaan. Komponentti käyttää toisen komponentin osoitinta voidakseen lukea ja muuttaa muuttujan arvoja suoraan tai käyttää aksessorifunktioita. Tällä tavalla komponentit ovat kokonaan riippuvaisia toisistaan, mutta toteutustapa on yksinkertaisempi ja suorituskykyisempi kuin toinen seuraava vaihtoehto. Epäsuora viestintä perustuu komponenttien lähettämiin viestipaketteihin, joita hallitsee komponentin omistava entiteetti. Komponentit lukevat viestejä omassa päivitysmetodissaan ja tulkitsevat niitä miten haluavat. Viestintätapa on joustavampi verrattuna suoraan kommunikaatioon ja antaa komponentin itse päättää omista asioistansa, mutta sen ohjelmointi ja vianetsintä on monimutkaisempaa ja viestien käsittely saattaa olla hitaampaa. (Crombecq, K. 2011.)

Komponenttien välinen viestintä voidaan unohtaa hyödyntämällä järjestelmäpohjaista implementaatiota, jossa komponentti on vain entiteetin ominaisuuksien kuvaaja ja datan säilyttäjä. Kaikki looginen toiminnallisuus on sen sijaan järjestelmissä ja järjestelmät tuntevat aina niiden vaatimat komponentit datan käsittelyä varten. Järjestelmät tavallaan hoitavat täten komponenttien välisen keskustelemisen niiden puolesta. Tällainen ohjelmointitapa on jous-

tavampi, sillä tietty järjestelmä ei tee mitään, ellei kaikkia sen vaatimia komponentteja löydy entiteetiltä. (Boreal Games 2013; Pie21 2011.)

Järjestelmäpohjainen toteutustapa voidaan ajatella yhden kerroksen lisäämisenä entiteetti-komponenttiarkkitehtuuriin: datakomponentit ja loogiset komponentit, eli järjestelmät jotka kirjoittavat ja lukevat datakomponentteja. (Pie21 2011.)

2.6 Valmiita toteutuksia

Komponenttipohjaista pelinkehittämistä helpottamaan on valmiita vapaasti käytettäviä toteutuksia. Valmis toteutus onkin hyvä tapa päästä alkuun ilman, että täytyy käyttää kymmeniä tunteja sellaisen itse ohjelmoimiseen.

2.6.1 Artemis Entity Framework

Artemis Entity Framework on Javalla ohjelmoitu komponenttipohjaiseen arkkitehtuuriin perustuva, pelikehitykseen suunniteltu avoimen lähdekoodin kirjasto. Artemis Framework -toteutus perustuu ideaan, jossa kaikki entiteetit ovat vain tunnisteita pelimaailmassa. Komponentit säilyttävät vain dataa ja järjestelmät käsittelevät entiteetit niiden piirteiden mukaan. (Gamadu 2012.)

Artemis on käännetty myös usealle eri kielelle, joihin sisältyvät muun muassa C++, C# ja Python. Se on tiettävästi suosituin ja levinnein entity system framework jakelussa. Lisää Artemiksen omalta sivulta: http://gamadu.com/artemis/ (Gamadu 2012.)

2.6.2 Cistron

Cistron on Karel Crombecqin C++:lla ohjelmoima komponenttipohjainen avoimen lähdekoodin kirjasto, joka on tarkoitettu erityisesti pelinkehitystä varten. Cistron toteuttaa pelimaailman objektit komponenttien kokoelmina, joissa komponentit keskustelevat keskenään

viestittämällä. Komponentit voivat ottaa viestejä vastaan myös ulkopuolelta niiden komentamista varten. (Crombecq, K. 2011.)

Cistron on suunniteltu olemaan joustava ja kevyt myös niihin käyttötarkoituksiin, joihin muut vaihtoehdot eivät sovellu hitautensa takia. Kirjasto on riippuvainen ainoastaan C++ standardikirjastosta, joten sen sisällyttäminen projektiin pitäisi olla helppoa. Lisää Cistronin sivulta: https://code.google.com/p/cistron/ (Crombecq, K. 2011.)

2.6.3 Ash Entity Framework

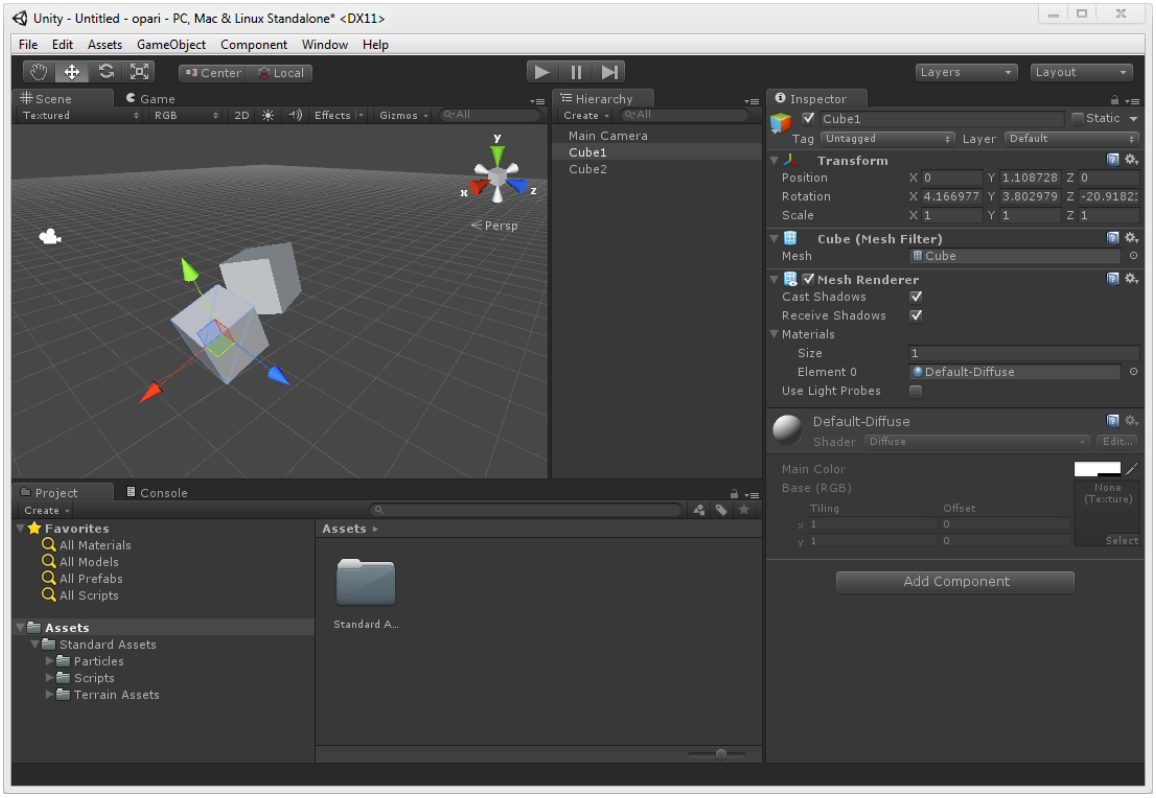
Ash Entity Framework on suorituskykyinen web-pelien kehittämiseen soveltuva kirjasto. Se on ohjelmoitu ActionScript 3:lla (Adobe Flash), ja se on käännetty muutamalle kielelle, joihin sisältyvät Javascript, Haxe ja Objective-C. (Lord, R. 2011.)

Kirjasto jakaa datan ja toiminnallisuuden samaan tapaan kuin Artemis, jossa komponentit ovat vain dataa varten ja funktionaalisuus on jätetty järjestelmille. Lisää Ashin sivulta: http://www.ashframework.org/ (Lord, R. 2011.)

2.6.4 Unity Game Engine

Unity on alustariippumaton suljetun lähdekoodin pelimoottori editorilla, joka tekee sillä kehittämisestä helpompaa aloitteleville pelin kehittäjille. Unityllä on 45 %:n markkinaosuus ja yli 3,3 miljoonaa rekisteröitynyttä pelinkehittäjää. (Unity Technologies 2014.)

Kuvio 5 näyttää Unity-editorin ja oikean reunan komponenttipaneelin (Inspector), joka mahdollistaa komponenttien parametrien muokkaamisen graafisessa editorissa. Unity-peliobjektit ovat tyhjiä entiteettejä, joille voidaan lisätä eri komponentteja. Jos valmiit komponentit eivät riitä, kehittäjä voi myös skriptaamalla luoda uutta toiminnallisuutta ja komponentteja. Unity tukee C#-, UnityScript- (räätälöity JavaScript versio) ja Boo-kieliä skriptaamista varten. (Unity Technologies 2014.)



Kuvio 5. Unity-editori ja objektien komponenttipaneeli (Unity Technologies, 2014.)

Unity on ilmainen testikäyttöä ja ilmaispelien kehitystä varten. Kaupallisten pelien kehitys ja lisäominaisuuksien avaus vaatii erillisen maksullisen lisenssin. Lisää Unityn sivulta: http://unity3d.com/ (Unity Technologies 2014.)

3 KOMPONENTTIPOHJAISEN ARKKITEHTUURIN TOTEUTUS

Tämän opinnäytetyön aikana ohjelmoin yksinkertaisen toteutuksen komponenttipohjaisesta arkkitehtuurista. Se on ohjelmoitu C++-ohjelmointikielellä, joka tekee siitä mahdollisen kääntää useammalle eri alustalle. Kieli soveltuu nopeutensa ansiosta myös hyvin pelinkehittämistä varten.

3.1 Toiminnallisuus

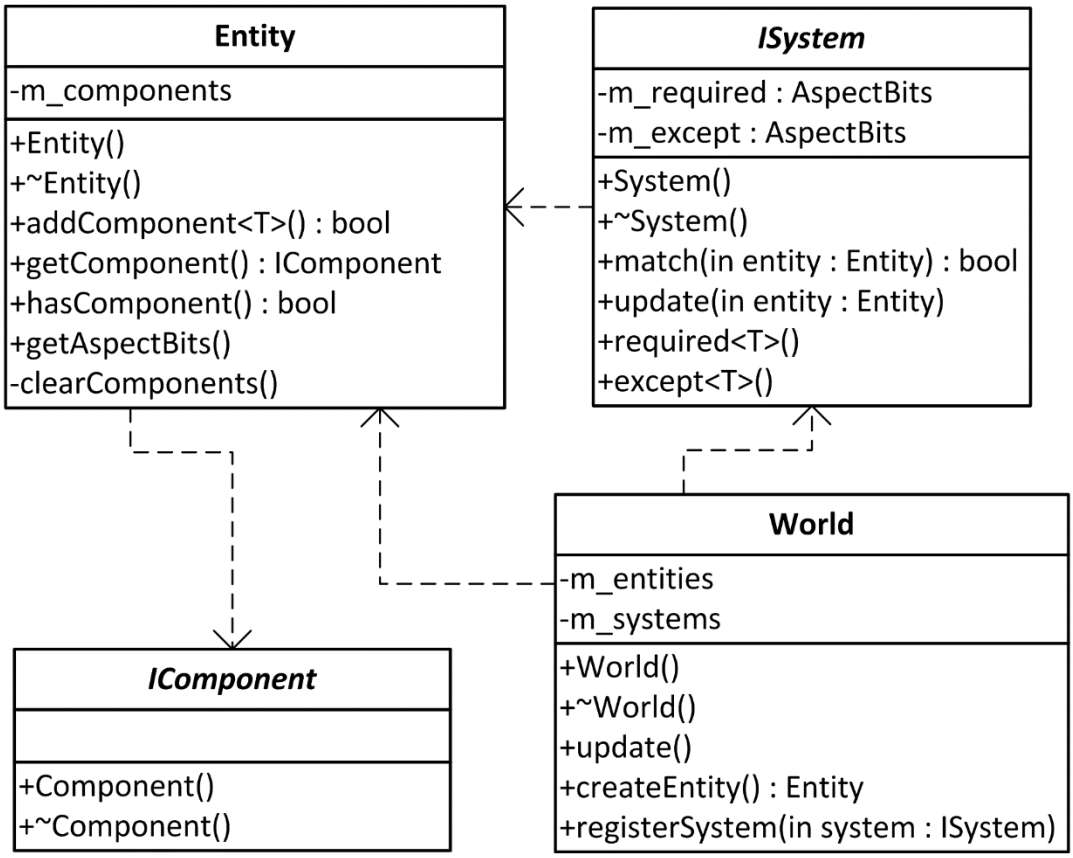
Tavoite on toteuttaa loogisen toiminnallisuuden implementoiviin järjestelmiin perustuva framework, joka käyttää komponentteja vain datan säilyttämiseen ja objektin piirteiden kuvailuun. Entiteetit ovat oma luokkansa ja hallitsevat omia komponenttejaan. Tämä on helpompi toteutustapa kuin vain tunnistenumeroiden käyttö, mutta käyttää hieman enemmän resursseja.

Toteutus käyttää samankaltaisia ideologioita, joita myös edellä esitelty Artemis Framework toteuttaa. Oman toteutuksen tulisi täyttää seuraavat ehdot:

1. Entiteetit, komponentit ja järjestelmät ovat kaikki erillisiä osia.
2. Entiteetti toimii komponenttien haltijana eikä sen toteutus riipu sen omistamista komponenteista.
3. Komponentti on vain datan säilytystä varten, eikä sillä ole muita metodeita kuin mitä oman datan muokkaamiseen vaaditaan.
4. Järjestelmät toteuttavat loogisen toiminnallisuuden käyttäen yhden tai useampien komponenttien dataa.
5. Maailma on entiteettien ja järjestelmien haltija, ja sen kautta voidaan luoda ja päivittää entiteettejä ja järjestelmiä.

3.2 Suunnittelu

Kuviossa 6 kuvataan järjestelmän eri osat ja niiden väliset riippuvuussuhteet.



Kuvio 6. Eri osien riippuvuussuhteet ja UML-kaaviot.

Maailmaluokka (World) on tavallaan tämän komponenttipohjaisen toteutuksen ydin, joka hallitsee järjestelmiä (ISystem) ja entiteettejä (Entity). Maailmaluokka toimii myös ns. tehdasluokkana uusille entiteeteille. Entiteettiluokka itse hoitaa omat komponenttinsa (IComponent). Kuviossa esitelty Component-luokka on ainoastaan periyttämistä ja polymorfismia varten tarkoitettu pohjaluokka.

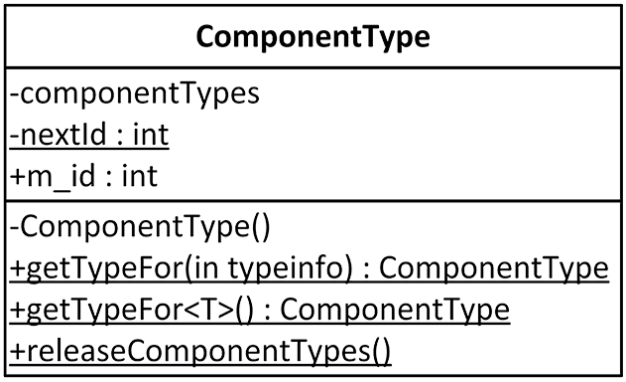
Järjestelmäluokka on myös perintään tarkoitettu pohjaluokka, ja periytetyt luokat toteuttavat järjestelmän päivitysmetodin, esimerkiksi MovementSystem voi toteuttaa sijainnin päivittämisen.

Järjestelmissä toteutetaan myös tarkastus, siitä onko entiteeteillä kaikki vaaditut komponentit. Tämä komponenttityyppien lista alustetaan järjestelmän luonnin yhteydessä. Pelimaailmaa

päivittäessä suoritetaan varsinainen tarkistus, jossa verrataan järjestelmien vaatimuksia olemassa oleviin entiteetteihin ja päivitetään vain ne entiteetit ja komponentit, jotka läpäisevät tarkistuksen. Tarkistus hyödyntää ComponentType-luokan indeksoituja tyyppejä.

Lisäksi toteutuksen tekoon luodaan useampia erilaisia spesialisoituneita apuluokkia.

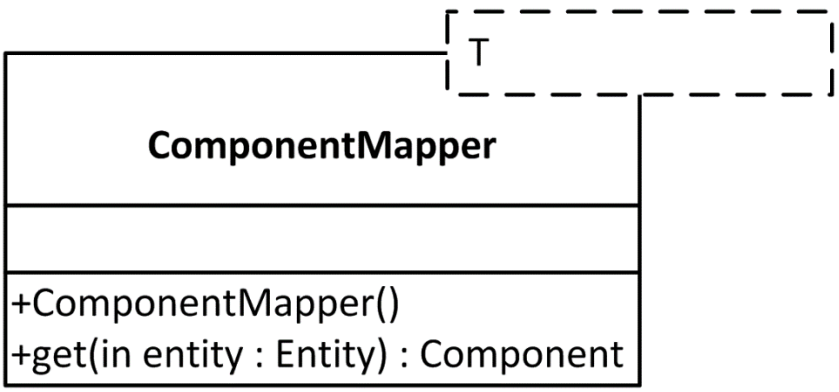
Entiteettiluokka hyödyntää komponenttien tehokkaaseen indeksointiin ja hakuun tarkoitettua ComponentType–apuluokkaa (kuvio 7), joka indeksoi eri komponenttityypit.



Kuvio 7. ComponentType–luokan UML-kaavio.

Tämän luokan avulla voidaan nopeasti hakea indeksoituja komponentteja C++-standardikirjaston listasta. Luokalla on staattisia metodeita eri tyyppien hakuun, ja se samalla indeksoi uudet tuntemattomat komponenttien tyypit ja palauttavat niille uniikin indeksin, jota muut luokat voivat sitten hyödyntää.

Komponenttien parametrien muokkaamiseen hyödynnetään yksinkertaista geneeristä ComponentMapper-luokkaa (kuvio 8).



Kuvio 8. ComponentMapper-luokan UML-kaavio.

ComponentMapper-luokka antaa järjestelmille pääsyn entiteetin komponenttien julkiseen käyttöliittymään hakemalla niiden osoittimen. Järjestelmän käyttämistä komponenteista jokaiselle alustetaan oma ComponentMapper-olio, jota järjestelmä voi käyttää päivitysmetodissaan.

3.3 Eri luokkien tekninen toteutus

Entity

Entity-luokka toteuttaa useampia C++ template-metodeita: addComponent, getComponent ja hasComponent. Ohjelmoija syöttää addComponent-metodille uuden osoittimen new-avainsanalla luotuun komponenttiolioon. Tämä metodi sitten hakee komponentin tyypin mukaan ComponentType-luokalta sille indeksin ja lisää sen hallitsemaansa std::map-listaan sekä päivittää entiteetin komponenttibitit. Näitä bittejä käytetään tarkistaessa, onko entiteetillä vaaditut komponentit, ja ne voidaan hakea julkiselta getAspectBits-metodilta. Nimensä mukaisesti getComponent palauttaa halutun komponenttityypin komponentin osoittimen entiteetiltä ja hasComponent palauttaa totuusarvon siitä, onko entiteetillä olemassa määriteltyä komponenttia.

IComponent

Tämä luokka on ainoastaan periyttämistä ja polymorfismia varten, eikä se toteuta erikseen mitään toiminnallisuutta.

Ohjelmoija voi periä omille komponenttiluokilleen IComponent-luokan luodessaan uusia komponenttityyppejä.

ISystem

Tällä periytykseen tarkoitetulla luokalla on yksi abstrakti metodi nimeltään update, jonka perivät luokat toteuttavat. Update-metodissa päivitetään järjestelmän logiikka, kun sitä kutsutaan päivittäessä maailmaa. Järjestelmäluokalla on myös kaksi template-metodia: required ja

except. Näillä ohjelmoija voi järjestelmää alustaessa määrittää, mitä entiteetin komponentteja järjestelmä tukee ja mitä ei entiteetiltä saa löytyä. Varsinainen tarkistus suoritetaan matchBits-metodissa, joka vertaa std::bitset-bittimaskeja keskenään. Bittimaskin pituus rajoittaa ohjelman komponenttien maksimimäärän. Oletusarvoisesti tässä toteutuksessa maskit ovat 32-bittisiä, mutta niiden pituutta voidaan helposti muuttaa, jos ohjelma vaatii useampia komponenttityyppejä.

Uusia järjestelmiä luodessaan ohjelmoijan uuden luokan tulee periä ISystem-luokka. Uudessa luokassa pitää määrittää järjestelmän vaatimat komponentit luokan constructor-metodissa ja luokalla tulee olla update-metodi päivystä varten. Komponenttien datan käsittelyä varten voidaan käyttää ComponentMapper-apuluokkaa. Kaikkia tarvittavia komponenttityyppejä varten alustetaan oma instanssi.

World

Pelimaailmaa ja kokonaisuutta hallitsemaan tarkoitettu maailmaluokka sitoo eri osa-alueet yhteen. World-luokka voi luoda uusia entiteettejä createEntity-metodillaan ja rekisteröi järjestelmät päivitykseen registerSystem metodilla. Molempia säilytetään niille tarkoitetuissa std::vector-listoissa. Pelin pääsilmukassa kutsutaan maailman update-metodia, joka sitten vastaavasti hoitaa kaikkien järjestelmien päivittämisen entiteeteille. Pelimaailmasta tulee alustaa vain yksi oma instanssi hallinnoimaan entiteettejä ja järjestelmiä ohjelman alussa.

ComponentType

Komponenttien tyyppien indeksointiin tarkoitettu luokka eroaa toteutuksessaan muista luokista. Luokan constructor-metodi on määritelty yksityiseksi, eikä ohjelmoijan kuulu erikseen instansoida luokasta olioita. Sen sijaan luokkaa käytetään kutsumalla sen staattista template-metodia getTypeFor, joka palauttaa syötetylle komponentin tyypille uuden tai jo aiemmin määritellyn indeksin. Indekseinä toimivat luokan itse instansoimat olio-osoittimet, joista jokaisella on uniikki kokonaisluku *m_id* ja bitti *m_bit*. Luokalla on myös staattinen muuttuja, jossa säilytetään seuraavaa indeksiä ja uutta indeksiä luotaessa seuraavan lukuarvoa kasvatetaan.

ComponentMapper

ComponentMapper on template luokka, joka on tarkoitus alustaa jokaiselle järjestelmän tukemalle komponenttityypille. Alustettua instanssia voidaan käyttää järjestelmän päivitysmetodissa hakemaan entiteetin komponentin osoitin käyttämällä luokan get-metodia, jolle annetaan parametrina referenssi entiteetistä. Tunnetulla komponenttityypillä alustettu ComponentMapper osaa sitten hakea entiteetiltä oikean tyypin komponentin.

3.4 Testaus ja käyttöesimerkki

Oman toteutukseni koodin testaamisen suoritin luomalla testijärjestelmän ja komponentteja, joita päivitin pelin pääsilmukkaa vastaavassa silmukassa. Listauksessa 2 esitetään testiohjelmani lähdekoodi. Ensimmäisenä alustetaan maailmasta instanssi ja rekisteröidään sille järjestelmä. Seuraavana luodaan uusi entiteetti ja lisätään sen komponentit, joille annetaan niiden oletusarvot samalla. Positiokomponentin osoitinta tarvitaan päivitetyn sijainnin tulostamista varten. Varsinaisessa pääsilmukassa päivitetään maailmaa, joka taustalla kutsuu järjestelmien päivitysmetodia ja päivittää positiokomponentin arvoja nopeuskomponentin mukaan. Tämän jälkeen tulostetaan päivitetty sijainti konsoliin nähtäväksi.

```
#include <iostream>
#include <memory>
#include <EntitySystem.hpp>
#include <Systems/MovementSystem.hpp>

int main()
{
    // Alustetaan maailmanmanageri
    std::unique_ptr<es::World> world(new es::World);

    // Rekisteröidään liikejärjestelmä
    world->registerSystem(new es::MovementSystem);

    // Luodaan uusi entiteetti ja annetaan sille komponentteja
    // Alustetaan komponenteille samalla oletusarvot
    es::EntityPtr entity = world->createEntity();
    entity->addComponent(new es::PositionComponent(-100.f, -50.f, 10.f));
    entity->addComponent(new es::VelocityComponent(-6.f, 3.f, 0.5f));

    // Haetaan positiokomponentin osoitin
    es::PositionComponent* position = entity->getComponent<es::PositionComponent>();

    // Pääsilmukka jossa hoidetaan päivitys
    while(true)
    {
        // Maailman päivitys, päivittää kaikki järjestelmät
        world->update();

        // Tulostetaan päivitetty sijaintiarvo
        std::cout << "Updated position: "
                  << position->x << ", "
                  << position->y << ", "
                  << position->z << "\n";
    }
}
```

Listaus 2. Testiohjelman lähdekoodi.

Listauksessa 3 näytetään esimerkki edellisen esimerkin liikejärjestelmäluokan toteutuksesta. Luokassa esitellään luokan constructor ja päivitysmetodi sekä alustetaan ComponentMapper käytetyille komponenteille. Järjestelmän vaatimat komponenttityypit alustetaan constructor-metodissa. Päivitysmetodissa päivitetään sijaintikomponentin arvoja lisäämällä niihin nopeuskomponentin vastaavan akselin arvo. Komponenttien arvoja päästään muokkaamaan käyttämällä aiemmin alustettuja ComponentMapper-olioita, joille syötetään referenssinä sen hetkinen entiteetti, jota järjestelmä päivittää.

```
#include <Core/System.hpp>
#include <Components/PositionComponent.hpp>
#include <Components/VelocityComponent.hpp>

namespace es {

class MovementSystem : public System
{
public:

    // Constructor jossa alustetaan vaatimukset
    MovementSystem(void);

    // Järjestelmän päivitysmetodi
    void update(Entity&);

protected:

    // Käytetään komponenttien arvojen muokkaamiseen
    ComponentMapper<PositionComponent> position;
    ComponentMapper<VelocityComponent> velocity;

};

MovementSystem::MovementSystem(void)
{
    // Tämä järjestelmä vaatii nämä komponentit
    required<PositionComponent>();
    required<VelocityComponent>();
}

void MovementSystem::update(Entity& e)
{
    // Päivitetään sijaintiarvot nopeuden mukaan
    // ComponentMapperille annetaan entity-referenssi
    position.get(e)->x += velocity.get(e)->x;
    position.get(e)->y += velocity.get(e)->y;
    position.get(e)->z += velocity.get(e)->z;
}

}
```

Listaus 3. Liikejärjestelmän luokan lähdekoodi.

Järjestelmillä ei ole määritelty rajaa minkälaista funktionaalisuutta ne toteuttavat. Toisin kuin komponentit, joiden pääasiallinen tarkoitus on olla vain datan säilytystä varten, järjestelmät voivat vapaasti toteuttaa useampia erilaisia metodeita.

Listauksessa 4 esitetään sijaintikomponentin lähdekoodi. Yleensä komponentit eivät ole kovin monimutkaisia eikä komponentilla välttämättä tarvitse olla ainuttakaan jäsenmetodia tai muuttujia. Sellaisia komponentteja voidaan käyttää merkkaamaan jotakin ominaisuutta, esimerkiksi pelaajan hahmoa tai pelin kameraa. Pääsääntöisesti komponentin tavoitteena on toimia vain dataa varten, joten komponenttien metodit tulisivat olla vain yksinkertaiseen kä-

sittelyyn liittyviä, kuten get- tai set-metodeita, ja varsinainen looginen funktionaalisuus toteutetaan järjestelmissä.

```
#include <Core/Component.hpp>

namespace es {

class PositionComponent : public Component
{
public:
    PositionComponent(void) :
        x(0.f), y(0.f), z(0.f) {}

    PositionComponent(float x, float y, float z) :
        x(x), y(y), z(z) {}

    void setPosition(float x, float y, float z)
    {
        this->x = x;
        this->y = y;
        this->z = z;
    }

    float x, y, z;
};

}
```

Listaus 4. Yksinkertaisen komponentin lähdekoodi, tässä tapauksessa sijaintikomponentti.

Uusia ja erilaisia komponentti- ja järjestelmäluokkia luomalla ohjelmoija voikin sitten kehittää mitä tahansa hän peliprojektiin tarvitseekaan. Osien modulaarisuus tekee kokonaisuuden skaalaamisesta isompaankin projektiin helppoa.

.

4 POHDINTA

Opinnäytteen tavoite oli esitellä komponenttipohjaisen objektinhallinnan perusteita pelinkehityksen näkökulmasta, ainakin alustavassa määrin. Teoriaosuudessa mielestäni onnistuin selittämään periaatteen ja syyt tällaisen ratkaisun käyttöön, joskin eri toteutustapojen selkeä ja kunnollinen erottelu on mielestäni vaikeaa.

Alussa olin kahden vaiheilla aiheen suhteen, että kirjoittaisinko opinnäytetyön pelimoottoreista. Kirjoitin kuitenkin jo aiemmin komponenttijärjestelmistä ajankohtaisseminaaria varten, joten opinnäytetyön rajoitus samaan aihealueeseen olikin varmaan järkevä valinta. Komponenttijärjestelmät ovat silti osa nykyaikaisia pelimoottoreita, joten opinnäytetyöni aihealue sivuaa pelimoottoreita, vaikken niistä erikseen mainitsekaan.

Ennen opinnäytetyöprosessia olin jo ohjelmoinut muutamia komponenttijärjestelmiä aiempiin peliprojekteihini, joskin ne erosivat opinnäytetyössä esittelemästäni teoriasta. Tekemäni toteutus tätä opinnäytetyötä varten ei ole kovin monimutkainen, enkä ole varma voiko sitä hyödyntää vielä sellaisenaan missään varsinaisessa peliprojektissa. Se toimii niin kuin sen kuuluisikin niiden testien mukaan mitä sille tein, mutta todellinen kokeilu olisi vasta se kun sitä oikeasti yrittää käyttää johonkin oikeaan peliprojektiin.

Opinnäytetyötä tehdessäni olen oppinut vielä lisää tästä aiheesta ja oppimani tieto ja ohjelmointikokemus tulee todennäköisesti olemaan hyödyllinen myös tulevaisuudessa.

LÄHTEET

Porter, Nicolas. 2012: Component-based game object system. Saatavilla: https://raw.githubusercontent.com/surjikal/cbgos-experiment/master/doc/nicolasporter-cbgos-paper.pdf (Luettu 11.9.2014)

Nimimerkki Pie21. 2011: Entity-Component Primer. Saatavilla: http://piemaster.net/2011/07/entity-component-primer (Luettu 11.9.2014)

Crombecq, Karel. 2011: Component-based programming. Saatavilla: http://www.codeximperium.be/stuff/Component-based%20programming.pdf (Luettu 11.9.2014)

Boreal Games. 2013: Understanding Component-Entity-Systems. Saatavilla: http://www.gamedev.net/page/resources/_/technical/game-programming/understanding-component-entity-systems-r3013 (Luettu: 11.9.2014)

Gamadu. 2012: Artemis Entity System Framework. Saatavilla: http://gamadu.com/artemis/index.html (Luettu 16.9.2014)

Martin, Adam. 2007: Entity Systems are the future of MMOG development. Saatavilla: http://t-machine.org/index.php/2007/09/03/entity-systems-are-the-future-of-mmog-development-part-1/ (Luettu 16.9.2014)

Unity Technologies 2014. Unity Game Engine. Saatavilla: http://unity3d.com/ (Luettu 6.10.2014)

Unity Technologies 2014. Unity Quick Facts. Saatavilla: http://unity3d.com/public-relations (Luettu 6.10.2014)

Lord, Richard 2011. Ash Entity Framework. Saatavilla: http://www.ashframework.org/ (Luettu 10.10.2014)