OAMK
OULUN AMMATTIKORKEAKOULU

Samuli Lehto

ENTITY-COMPONENT-SYSTEM-OHJELMISTOARKKITEHTUURI PELIKEHITYKSESSÄ

ENTITY-COMPONENT-SYSTEM-OHJELMISTOARKKITEHTUURI PELIKEHITYKSESSÄ

Samuli Lehto
Opinnäytetyö
Kevät 2021
Tietotekniikan tutkinto-ohjelma
Oulun ammattikorkeakoulu

TIIVISTELMÄ

Oulun ammattikorkeakoulu
Tietotekniikan tutkinto-ohjelma, ohjelmoinnin suuntautumisvaihtoehto

---

Tekijä: Samuli Lehto
Opinnäytetyön nimi suomeksi: Entity-component-system-ohjelmistoarkkitehtuuri pelikehityksessä
Työn ohjaajat: Teemu Korpela, Tuula Hopeavuori
Työn valmistumislukukausi ja -vuosi: Kevät 2021
Sivumäärä: 27

Opinnäytetyön aihe on entity-component-system-ohjelmistoarkkitehtuuri ja sen soveltuminen peliohjelmistokehitykseen. Opinnäytetyössä kuvataan ECS-arkkitehtuuri ja kerrotaan sen hyödyistä ja toteuttamisesta. Tavoitteena oli selvittää, soveltuuko arkkitehtuuri ohjelmistokehitykseen ja pitävätkö sen esitetyt hyödyt paikkaansa.

ECS-arkkitehtuurin osaset määriteltiin ohjelmistoalan ammattilaisten kokemusten pohjalta. Arkkitehtuuria käyttäen suunniteltiin ja ohjelmoitiin esittelypeli, jonka etenemisen avulla arvioitiin arkkitehtuurin soveltuvuutta ja esitettyjä hyötyjä.

Esittelypeli valmistui suunnitelman mukaan täyttäen vaatimukset. Sen tekemisestä saatujen kokemusten pohjalta voidaan todeta, että ECS-arkkitehtuuri on varteenotettava tekniikka tietokonepelin tai muun interaktiivisen ohjelman tekoon.

Asiasanat: peliohjelmointi, ohjelmistotuotanto, ohjelmistoarkkitehtuuri

ABSTRACT

Oulu University of Applied Sciences
Degree Programme in Information and Communication Technology, Software Development

---

Author: Samuli Lehto
Title of thesis: Entity-Component-System Architecture in Game Development
Supervisors: Teemu Korpela, Tuula Hopeavuori
Term and year when the thesis was submitted: Spring 2021
Pages: 27

Game development is difficult and entity-component-system architecture was created to manage the challenges of game development. The objective of the thesis is to define the ECS architecture, to study reasons it is used, and how could a program be made using it.

The thesis describes the ECS architecture. The theory is then tested by designing and writing a demo program using the architecture. A chapter details the experiences of using the architecture.

The demo program was successful, and the experiences gained programming it demonstrated that the ECS architecture is good for creating interactive software like computer games. The demo program is relatively simple and small in scope and thus it could not be used to test if significant performance gains could be gotten by making a program with entity-component-system architecture.

Keywords: game programming, software engineering, software architecture

SISÄLLYS

TIIVISTELMÄ 3
ABSTRACT 4
SISÄLLYS 5
1 JOHDANTO 6
2 ENTITY-COMPONENT-SYSTEM-OHJELMISTOARKKITEHTUURI 7
2.1 ECS-arkkitehtuurin osat 7
2.2 Hyödyt 8
2.2.1 Selkeys 8
2.2.2 Joustavuus 9
2.2.3 Tehokkuus 9
2.2.4 Testattavuus 10
2.3 Käyttökohteet 10
3 ECS-ARKKITEHTUURIN TOTEUTUSTAPOJA 12
3.1 Entity-osa 12
3.2 Component-osa 12
3.3 Componentien säilytystapoja 14
3.4 System-osa 15
4 ESITTELYOHJELMA 17
4.1 Vaatimukset 17
4.2 Teknologiat 18
4.3 ECS-arkkitehtuuri esittelyohjelmassa ja toteutusyksityiskohdat 18
4.4 Kokemukset esittelyohjelman toteutuksesta ECS-arkkitehtuurilla 22
5 YHTEENVETO 24
LÄHTEET 25

1 JOHDANTO

Videopelien kehitys on vaativa tehtävä. Pelikehityksen ongelmia ovat projektin aikataulutus, muuttuvat toiminnallisuusvaatimukset ja tiukat tehovaatimukset. Tämä ympäristö vaatii peliohjelmalta ohjelmistoarkkitehtuuria, joka on joustava muokata, mutta tehokas suorituskyvyltään. Entity-component-system-arkkitehtuuri kehitettiin ratkaisemaan näitä pelikehityksen ongelmia. (1.)

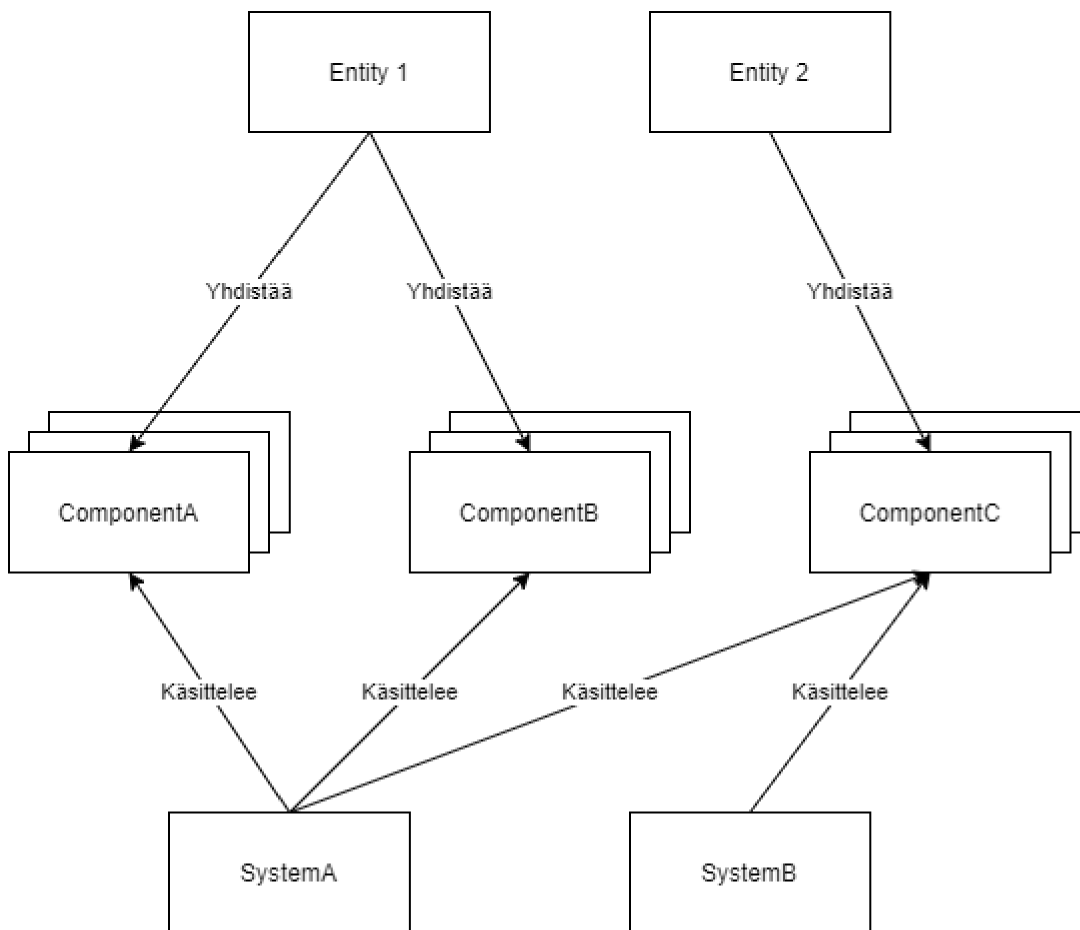
ECS-arkkitehtuurin kolme pääkäsitettä ovat entity, component ja system. Entity on yksi dynaamisesti koottu ohjelman konkreettinen osanen, kuten pelaajahahmo tai käyttöliittymän nappula. Componentit sisältävät kaiken ohjelman datan, ovat aina liitettyinä entiteetteihin ja antavat näille ominaisuuksia. Systemit ovat ohjelman logiikka ja kaikki ohjelman toiminnallisuudet erotetaan niihin. Systemit toimivat muokkaamalla componentien dataa. (2; 3.) ECS-arkkitehtuurin etuja ovat joustavuus, testattavuus ja tehokkuus (1; 4). Arkkitehtuuria käytetään erityisesti videopelikehityksessä.

Ensimmäisiä kokeiluja siirtyä olio-ohjelmoinnin luokkahierarkiasta joustavampaan järjestelmään oli Scott Bilasen pelioliojärjestelmä, jossa peliolio on vain tunnisteellinen tyhjä säiliö, johon laitetaan haluttuja komponentteja (5). Adam Martin kehitti ideaa esittelemällä blogissaan oman entiteettijärjestelmänsä sarjassa viestejä. Martin erotti toiminnallisuudet komponenteistä systemeiksi ja antoi arkkitehtuurin osasille nimet, jotka ovat käytössä nykyään (6).

Tämän opinnäytetyön tavoite on kuvailla ECS-arkkitehtuuri ja suunnitella ja toteuttaa ohjelma arkkitehtuuria käyttäen. Työn teoriaosassa määritellään arkkitehtuuri, sen osat ja rakenne, toteutusosassa esitellään toteutusvalintoja ja -yksityiskohtia ja pohdintaosassa kerrotaan kokemukset ohjelman tekemisestä ECS-arkkitehtuurilla.

2 ENTITY-COMPONENT-SYSTEM-OHJELMISTOARKKITEH-TUURI

Entity-component-system-arkkitehtuuri on ohjelmistoarkkitehtuuri, jossa ohjelman logiikka ja data erotetaan toisistaan (kuva 1). Ohjelman data sijoitetaan componenteihin ja logiikka systemeihin. Entity on yksi ohjelman osanen, johon kuuluu yksi tai useampia componenteja ja jokainen component kuuluu yhteen entityyn. (1.)



KUVA 1. ECS-arkkitehtuuri kaaviona

2.1 ECS-arkkitehtuurin osat

ECS-arkkitehtuurin component-osat säilyttävät ohjelman datan. Koska arkkitehtuurissa korostetaan tiedon ja toiminnallisuuksien erottamista, componenteilla ei ole mitään toiminnallisuuksia. Tieto jaetaan eri componenteihin käytön mukaan.

Samoissa toiminnoissa käytetyt tiedot laitetaan samaan componentiin. Componentit ovat aina liitoksessa yhteen entityyn ja antavat entitylle componentin ominaisuuden. (2; 3.) Component on atominen eli se sisältää pienimmän mahdollisimman määrän dataa, mitä jokin system tarvitsee toimiakseen eikä se jaa dataa yhdenkään muun componentin kanssa (7). Esimerkiksi entityllä, joka on pelihahmo, voi olla ikä-component, mikä kertoo ohjelmalle, että kyseinen entity altistuu pelin ikääntymismekaniikoille. Vastaavasti ikä-componentiton pelihahmo-entity voidaan tulkita iättömäksi.

Entityllä itsellään ei ole dataa eikä toiminnallisuuksia, vaan se on tunniste, joka yhdistää joukon componenteja kokonaisuudeksi (3). Entity on jokin ohjelman osanen (1; 2). Esimerkiksi graafisessa käyttöliittymässä painike on entity, jonka componentit kuvaavat painikkeen tiedot, kuten sen sijainnin, koon tai painallustilan. Yksinkertaisimmillaan entity on vain yksilöivä kokonaislukutunnus (3).

ECS-arkkitehtuurissa ohjelman data ja toimintalogiikka erotetaan toisistaan (1; 2). Entityt ja componentit ovat ohjelman data ja systemit sen logiikka. Kaikki ohjelman toiminnallisuudet ovat systemeissä. Koska kaikki ohjelman data säilytetään componenteissa, ovat systemit tilattomia eli ne eivät säilytä mitään tietoa (8). System toimii käymällä jatkuvasti läpi kaikki entityt, joilla on systemin toimintaan vaadittavat componentit, ja muuttamalla näiden componentien tietoja (3). Systemit ovat riippumattomia toisistaan ja kommunikoivat keskenään vain välillisesti muokkaamalla ja lukemalla componenteja, joista molemmat ovat kiinnostuneet (9).

2.2 Hyödyt

2.2.1 Selkeys

ECS-arkkitehtuurissa ohjelman data ja toiminnallisuudet erotetaan toisistaan eikä ohjelman tilaa piiloteta esimerkiksi olio-ohjelmoinnin näkyvyysalueiden keinoin eikä arkkitehtuurissa luoda riippuvuuksia systemien välille. Nämä ominaisuudet helpottavat ohjelmakoodin seuraamista. Ohjelman tilan tarkastelemiseksi voi-

daan vaivatta luoda ohjelman sisäinen virheenjäljitin (debugger) tai rekisteri datamuutoksia varten, joissa virheviestit tai lokitiedot ovat omat componentinsa ja niiden tulostuksesta vastaa tarkoitukseen tehty system. (4.)

Componentien oleminen entityssä tarkoittaa, että entityllä on näiden componentin tarjoamat ominaisuudet ja että componenteista kiinnostuneet systemit käsittelevät entityn suorituksessaan. Vastaavasti componentin olemattomuus tarkoittaa, että systemit eivät käsittele entityä. Tällä tekniikalla voidaan välttää tilan tarkastuksia, kuten ehtolausepuita, jotka vaikeuttavat ohjelman suorituksen seuraamista, systemin suorituksen aikana (10). Tätä tekniikkaa varten voidaan varta vasten luoda tyhjiä componenteja, joiden ainoa tarkoitus on merkitä entity systemeille (1).

2.2.2 Joustavuus

ECS-arkkitehtuurissa ohjelman osasia eli entityjä ei määritellä käännösaikana, vaan ne luodaan ajon aikana luomalla ja liittämällä componenteja niihin. Tämä tekee uudenlaisten entityjen luonnista ja olemassa olevien entityjen muokkaamisesta helppoa. Liittämällä eri kokoelma componenteja entityyn saadaan uudentyyppinen entity ja liittämällä tai poistamalla component entitystä tämän toiminta muuttuu, koska liitetyt componentit määrittävät, mille systemeille entityt altistuvat. (1; 2.)

2.2.3 Tehokkuus

ECS-arkkitehtuuri yksinkertaistaa rinnakkaislaskennan käyttämistä ohjelmassa (1). Jotta rinnakkaislaskentaa voidaan käyttää, suoritettava tehtävä on pystyttävä jakamaan toisistaan itsenäisiin alatehtäviin, minkä arkkitehtuurissa mahdollistaa datan jakaminen käytön mukaan atomisiin componenteihin ja toisistaan riippumattomat systemit (2). Rinnakkaislaskentaa voidaan käyttää kahdella tavalla: systemin sisäisen toiminnan jakaminen alatehtäviin tai itsenäisten usean systemin suorituksen muodostamien ketjujen rinnakkain suorittaminen (11).

Datan jakaminen componenteihin, jotka voidaan säilyttää yhtenäisellä muistialueella, ja kaikkien samantyyppisten componentien käsittely systemeissä kerralla on myös tehokas tapa käyttää nykyaikaisten suorittimien välimuistia (1; 4; 10).

Muistioptimoitu osaohjelma tai algoritmi voi olla moninkertaisesti nopeampi kuin muistin käytöstä välittämätön vastaava (10).

2.2.4 Testattavuus

ECS-arkkitehtuurilla toteutetun ohjelman testaaminen on helppoa. Koska testattava ohjelmalogiikka on toisistaan riippumattomissa systemeissä, testaaminen ei vaadi injektioita eikä sijaisluokkia tai -funktioita (mock class/function); ainoat riippuvuudet ovat testattavan systemin käsittelemät componentit. Testaaminen suoritetaan alustamalla testidata, suorittamalla testattava system ja tarkistamalla, että data on muuttunut oletetun mukaisesti. Yksikkö- ja integraatiotestien määrittely on yksinkertaista. System voidaan määritellä yksiköksi ja näin yksikkötestaaminen on yhden systemin testaamista. Integraatiotestaaminen on usean systemin muodostaman putken tai sarjan testaamista. (4.) Koska arkkitehtuurissa pyritään välttämään boolean-tyyppisistä lipuista aiheutuvia tiloja componentien olemassa- tai poissaolon avulla, ovat testit suoraviivaisempia kirjoittaa ja selvitään pienemmällä määrällä testejä (10).

2.3 Käyttökohteet

Vuorovaikutteisissa ohjelmissa käyttäjä tulee pitää tietoisena ohjelman tilasta esimerkiksi animaatioin ja käyttäjän syötteeseen pitää reagoida välittömästi. Tämän takia vuorovaikutteinen ohjelma on luonteeltaan syklinen eli ohjelma jatkuvasti laskee uudestaan oman tilansa käyttäjän syötteen ja edellisen tilan perusteella. ECS-arkkitehtuurin jatkuvasti componentit läpi iteroivat ja niitä muokkaavat systemit sopivat tähän käyttötarkoitukseen hyvin. (4.)

Pelit erityisesti ovat suosittu käyttökohde ECS-arkkitehtuurille (1). Datan ja toimintojen erottamisesta johtuva systemien välinen riippumattomuus on tärkeä esimerkiksi verkko- ja grafiikkatoimintojen ohjelmoimiseen (2; 6; 8). Arkkitehtuurin joustavuus sopii hyvin peliprojektien muuttuviin vaatimuksiin ja pelimaailman sisäisten monimutkaisten vuorovaikutusten hallintaan (1; 2). Pelit vaativat usein paljon suorituskykyä suorittavalta koneelta. Datan jakaminen ja järjestäminen

atomisiin yksiköihin eli componenteihin mahdollistaa muistioptimoinnin ja rinnakkaislaskennan, joilla voidaan saavuttaa merkittäviä suorituskykyparannuksia peleissä (2; 4; 12).

3 ECS-ARKKITEHTUURIN TOTEUTUSTAPOJA

3.1 Entity-osa

Entityn tehtävä on yksilöidä ja koota yhteen componentit, joten entityn on vähintään sisällettävä ainutlaatuinen tunniste, joka erottaa sen muista entityistä. Yleisesti suositellaan, että entity toteutetaan 32-bittisellä kokonaisluvulla (8; 13). Erityistapauksissa, kuten levylle tallennuksessa tai kommunikoimisissa verkon yli, voidaan käyttää UUID:tä (13).

Usein entityihin liittyy erilaista metatietoa, kuten mitä componenteja entityllä on tai mikä on entityn ihmiskielinen nimi ohjelman virheenjäljitystä varten. Siihen, miten tämä metatieto tallennetaan ohjelmassa, on erilaisia ratkaisuja. Martin Adam kieltää yhdistämästä entityn metatietoa entityn toteutukseen ja kehottaa sen sijaan luomaan uusia componenteja metatietoa varten (14). Myös lievempiä kantoja on olemassa. Overwatch-pelin ECS-arkkitehtuurissa entityn toteutus sisältää listan kaikkiin entityn componentien muistiosoitteisiin (8).

Entityillä ei ole olio-ohjelmoinnin luokkamaista käännösaikaista tyyppiä, joka kertoo entityn ominaisuudet, vaan entityt luodaan ajonaikaisesti kokoamalla mielivaltainen joukko componenteja. Kuitenkin usein ohjelma tarvitsee suuren määrän samankaltaisia osasia. Esimerkiksi graafinen käyttöliittymä voi tarvita paljon nappuloita tai tekstikenttiä. Jotta ohjelmoija voi helposti luoda paljon usein käytettyjä samanlaisia entityjä, voidaan määritellä arkkityyppi (archetype). Arkkityyppi on ennalta määritetty joukko componenteja ja sitä kutsuttaessa luodaan määritetyt componentit ja entity kokoomaan componentit. Arkkityyppi voidaan toteuttaa funktiolla. Arkkityyppi ei ole luokka tai merkittävä rakenne, vaan se on oikotie ohjelmoijalle. Arkkityypillä luotua entityä voidaan muokata ajon aikana sen luomisen jälkeen, kuten muitakin entityjä. (3; 14.)

3.2 Component-osa

Component toteutetaan ohjelmointikielen mukaan tietueella (struct), luokalla tai vastaavalla componentin tietotyypit yhteen kokoavalla rakenteella (11; 15). Componentit eivät sisällä pelilogiikkaa, mutta jossain toteutuksissa niille on määritetty

apufunktioita, kuten aksessoreita (3; 8). Mahdollisia tapoja toteuttaa entityn ja componentien välinen kokoomasuhde on esimerkiksi säilyttää component entityn tunnistelukua vastaavassa taulukon kohdassa tai pitää kirjaa componentien entityistä erillisellä hakutaululla, jossa yhdistetään entity tiettyyn componentiin (11; 14).

Ohjelman data jaetaan componenteihin käytön mukaan niin, että yhdessä component-tyypissä on pienin mahdollinen määrä dataa yhden ohjelman kannalta merkityksellisen ominaisuuden toteuttamiseksi (4; 7). Componentilla ei kuitenkaan tarvitse olla yhtäkään muuttujaa. Tällöin componentin kuuluminen entityyn on itsessään oleellinen tieto, joka kertoo componentista kiinnostuneelle systemille, mitä entityjä käsitellä (1). Componentin sisältämän tiedon ainutlaatuisuudesta on kahta katsantokantaa. Adam Martinin mukaan componentit eivät koskaan jaa dataa keskenään. Data on tietokantatermein normalisoitu (7). Normalisoinnilla pyritään välttämään datan toistoa ja takaamaan sen eheys (16). Toisaalta on useita tuotannossa olevia ohjelmia, joiden kehityksessä on huomattu, että suoritustehosyistä voi olla aiheellista kopioida dataa componentien välillä (10; 12). Datan kopioiminen toiseen componentiin voi myös yksinkertaistaa ja helpottaa suoritettavia operaatioita (10).

Kaikki samantyyppiset componentit säilytetään yhdessä tietorakenteessa riippumatta siitä, mihin entityyn ne kuuluvat, structure of arrays -tapaan (1; 8; 11). Tietorakenteen tulisi olla muistialueeltaan jatkuva, jos käytetty ohjelmointikieli mahdollistaa jatkuvan muistialueen varaamisen olio- tai tietuetyypeille, jotta suorittimen välimuisteja voitaisiin hyödyntää tehokkaasti (1; 11; 12). Tavalliset taulukkotyypit ovat yksinkertainen ja käytetty tietorakenne componentien säilyttämiseen (8; 11).

Pienillä entity-määrillä ohjelmassa myös component-taulukoiden koot pysyvät pieninä ja tiettyyn entityyn kuuluvan componentin hakeminen taulukosta lineaarisella haulla on tehokasta. Tämä johtuu suorittimen ennalta hakemisesta (prefetching) ja mahdollisesti taulukoitujen componentien läheisyydestä toisiinsa (data locality) (12; 17). Componentien määrän kasvaessa lineaarisen haun keskisuori-

tusaika kasvaa lineaarisesti, koska algoritmin aikakompleksisuus on O-notaatiolla esitettynä O(n) (18). Suurilla component-määrillä voi olla aiheellista toteuttaa yksittäisen componentin haku muulla tavalla, esimerkiksi vakioaikaisesti hakutaululla, jossa componentin tunnusluku toimii avaimena ja palautettu arvo on componentin osoitin tai sijainti taulukossa (15).

3.3 Componentien säilytystapoja

Datan säilytystapa on tärkeä, jos tavoitteena on ohjelman tehokas suoritus (17). Koska ECS-arkkitehtuurin esitetty hyöty on sen tehokkuus, esitellään pari tapaa säilyttää componentit ohjelmassa.

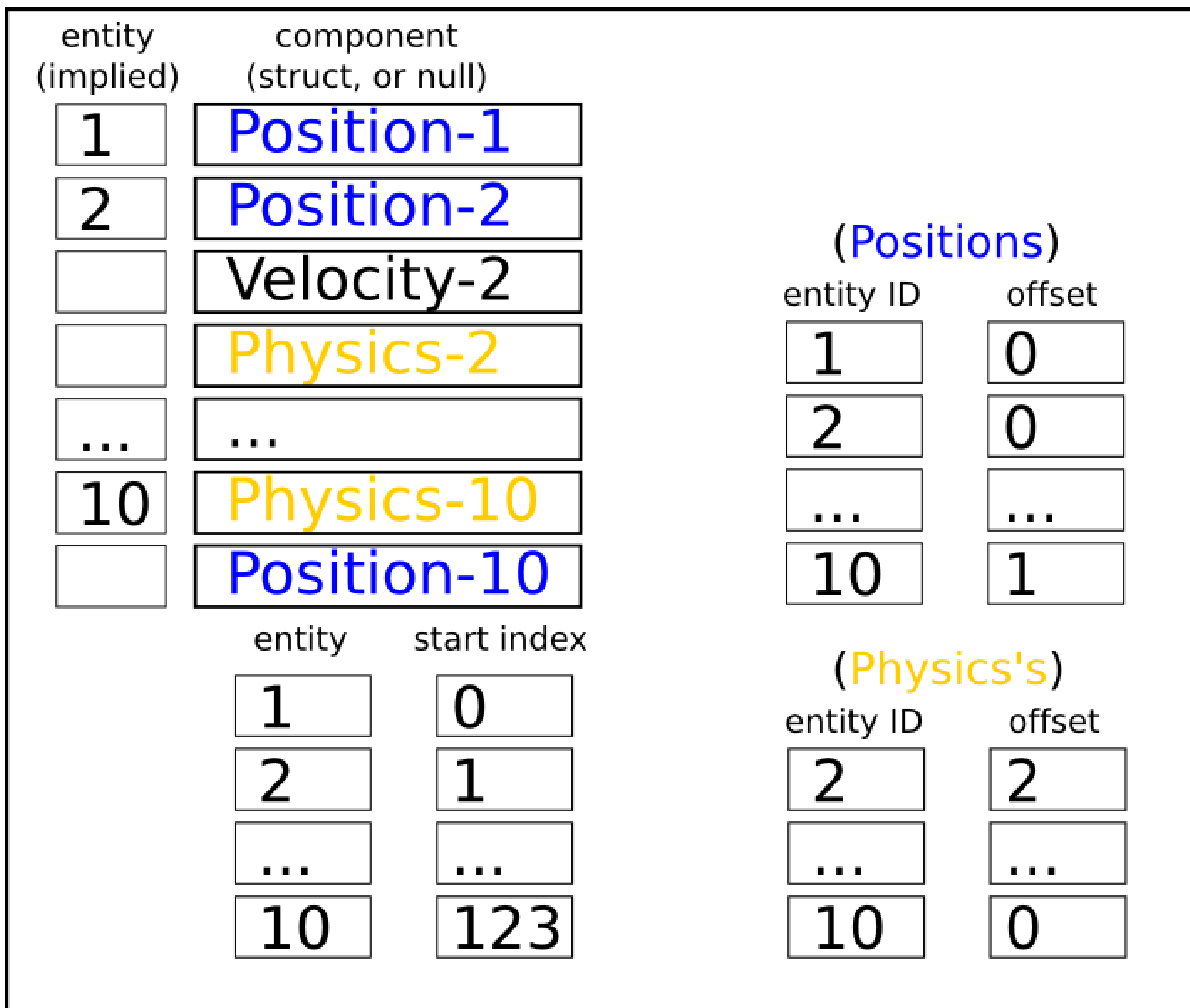
Ensimmäinen tapa on tallentaa kaikki samantyyppiset componentit yhteen isoon taulukkoon, jossa elementin indeksiluku vastaa componentin entityn tunnistelukua (kuva 2). Ratkaisun vahvuus on sen toteutuksen yksinkertaisuus. Heikkous on tehoton välimuistin käyttö. (9.) Väljästi täytetyssä taulukossa yhteen cache lineen osuu vähemmän componenteja kuin pakatussa taulukossa (17).

entity (implied)	component (struct, or null)	size (bytes used)
1	Position-1	64bits
2	(null)	64bits
3	(null)	64bits
4	Position-4	64bits
...	...	...

KUVA 2. Taulukko yhdelle component-tyypille (9)

Toinen tapa on megataulukko. Megataulukko-ratkaisussa componentit sijoitetaan yhteen isoon taulukkoon. Koska eri component-tyypit voivat olla eri kokoisia,

ei megataulukko ole varsinaisesti taulukko, vaan yksi yhtenäinen muistilohko, johon sijoitetaan component-dataa jossain järjestyksessä. Entityihin voidaan lisätä ja niistä voidaan poistaa componenteja ajon aikana, joten megataulukon rakenteesta tulee pitää kirjaa esimerkiksi hakutaulujen avulla. (kuva 3) Tämä on teknisesti vaativampi ratkaisu kuin oman taulukon antaminen jokaiselle componenttyypille, mutta mahdollistaa muistioptimoinnin, josta voi olla tehohyötyä. (9.)



KUVA 3. Megataulukko hakutauluineen (9)

3.4 System-osa

Koska system on toiminnallisuus ilman tilaa, systemin toteutus on yksinkertaisimmillaan funktio (1). Systemeitä ajetaan jatkuvasti ohjelman pääsilmukassa vasten niitä entityjä, joilla on kaikki systemin toimintaan vaadittavat componenit. Syste-

mit lukevat ja muokkaavat componenteja (2). Systemit eivät kutsu muita systemeitä, vaan kommunikointi systemien välillä tapahtuu välillisesti muokkaamalla ja lukemalla componenteja, joista molemmat ovat kiinnostuneet (9). Apufunktioita, kuten entityjen luontiin tarkoitettuja arkkityyppifunktioita, voidaan kuitenkin kutsua (8; 14). Ohjelman logiikka muodostuu systemien toiminnallisuuksista, mutta myös systemien keskinäisestä suoritusjärjestyksestä (1; 15).

Jossain toteutuksissa jokaista componentia kohden on yksi system, joka käy läpi vain sen tyyppiset componentit (2). Tällaista järjestelyä pidetään liian rajoittavana ja vaihtoehto on toteutus, jossa system voi toimia mielivaltaisilla syötteillä, jotka voivat olla eri component-tyyppejä tai muita tietotyyppejä (15).

Yhdestä componentistä kiinnostuneiden systemien ohjelmointi on yksinkertaista: iteroidaan tämän tyyppiset componentit systemissä ja tehdään tarvittavat luku- ja kirjoitusoperaatiot. Mutta usein systemit tarvitsevat useita componenteja toimiakseen ja ongelma on, kuinka löydetään ne entityt, joilla on kaikki systemin toimintaan tarvittavat componentit. Yksi tapa on pitää kirjaa kaikista entityiden omistamista componenteista ja ennen systemin suoritusta käydä läpi kaikki entityt ja listata ne, joilla on tarvittavat systemin toimintaan tarvittavat componentit. Esimerkiksi jokaisella entityllä voi olla bittitaulu, jossa yksittäinen bitti kertoo, onko entityllä tätä bittiä vastaava component vai ei, ja jokaisella systemillä on samankokoinen bittitaulu, joka kertoo, mitkä componentit system vaatii toimiakseen. Entityn yhteensopivuuden systemin kanssa voidaan tarkistaa lausekkeella:

$$(entityn\ bittitaulu\ \&\ systemin\ bittitaulu) == \ systemin\ bittitaulu$$

Lausekkeessa &-symboli on ja-bittioperaatio. Jos entityn bittitaulu olisi 0b00010110, entityllä olisi componentit 2, 3 ja 5. Jos haluaisimme tietää, onko entity yhteensopiva systemin kanssa, joka vaatii componentit 2 ja 3 eli jonka taulu on 0b00000110, suorittaisimme ensin ja-bittioperaation 0b00010110 & 0b00000110, jonka tulos on 0b00000110, ja sitten suorittaisimme yhtä suuri kuin -operaation, joka palauttaisi arvoksi toden eli esimerkin entity on yhteensopiva systemin kanssa. (11.)

4 ESITTELYOHJELMA

Opinnäytetyöhön tehtiin ohjelma ECS-arkkitehtuuria noudattaen tarkoituksena kokeilla arkkitehtuurin esitettyjä hyötyjä ja saadakseen kokemuksia ohjelman toteuttamisesta arkkitehtuurilla. Esittelyohjelmaksi valittiin tietokonepeli, koska peliohjelman ominaispiirteet sopivat hyvin arkkitehtuurin esitettyjen vahvuuksien kokeilemiseksi. Peleissä on paljon toistensa kanssa vuorovaikuttavia osasia, mikä vaatii arkkitehtuurilta joustavuutta, jonka arkkitehtuuri suo datan ja toiminnallisuuksien erottamisella. Arkkitehtuurin tehokkuuden testaaminen suunnitellulla esittelypelillä ei onnistu, koska työhön varattu aika ei riitä vaativan pelin kehittämiseen.

4.1 Vaatimukset

Esittelypelille määriteltiin vaatimuksia, joiden toteutumisen perusteella voidaan arvioida arkkitehtuuria.

Esittelypelin vaatimukset:

- Pelaaja voi liikuttaa näppäimistöllä määrättyä hahmoa pelimaailmassa.
- Pelaaja voi vuorovaikuttaa muiden näkyvien pelimaailman entityjen kanssa.
- Peli on ruutupohjainen eli kaikki pelimaailman sijainnit, liikkeet ja ulottuvuudet lasketaan ruuduissa. Esimerkiksi aarrearkku voi olla ruudussa koordinaatissa x = 4, y = 3, hahmo voi liikkua yhden ruudun ylöspäin, tulipallotaian räjähdys voi osua kaikkiin asioihin kolmen ruudun levyisen ja kolmen ruudun pituisen alueen sisällä. Pelin graafisen esityksen ei kuitenkaan tarvitse noudattaa tiukkaa ruutupohjaisuutta.
- Peli on vuoropohjainen eli jokainen pelaajan ja tekoälyn ohjaama asia suorittaa toimintoja alusta loppuun ilman, että muu ohjattu asia voi aloittaa toimintoa välissä. Pelin graafisen esityksen ei tarvitse olla vuoropohjainen. Esimerkiksi pelaajan ajan säästämiseksi kaikkien tekoälyn ohjaamien asioiden animaatiot voidaan näyttää samanaikaisesti.
- Pelaajahahmo animoidaan sprite sheet -tekniikalla.

- Pelissä on ainakin yksi tekoälyn ohjaama asia, joka liikkuu pelimaailmassa ja jonka kanssa pelaajahahmo voi olla vuorovaikutuksessa.
- Pelissä on tekstikenttä, joka pitää kirjaa pelin tapahtumista.

4.2 Teknologiat

Esittelypeli ohjelmoitiin C++-ohjelmointikielellä, koska se on entuudestaan tuttu työn tekijälle ja kieli mahdollistaa matalan tason muistinhallinnan, mikä voi tulla hyödyksi, jos halutaan tehdä malliksi suorittimen välimuistin kannalta tehokkaita tietorakenteita (19).

Graafisen interaktiivisen ohjelman matalatasoinen toteutus vie paljon aikaa eikä ole opinnäytetyön kannalta aiheellinen, joten graafisen esitysasun tekemiseen käytettiin SFML-kirjastoa, joka tarjoaa yksinkertaisen rajapinnan ohjelmaikkunan luontiin ja siihen piirtämiseen sekä käyttäjäsyötteen lukemiseen näppäimistöltä (20). Opinnäytetyön tekijällä on kokemusta kirjaston käytöstä, joten käyttöönotossa ja käyttämisen opettelussa ei kulunut paljon aikaa.

4.3 ECS-arkkitehtuuri esittelyohjelmassa ja toteutusyksityiskohdat

Entity esittelypelissä on 32-bittinen uniikki etumerkitön kokonaisluku (kuva 4). Jokaisella entityllä on sitä vastaava Signature-tietue, joka on 32-bittinen bittitaulukko, joka säilyttää tiedon entityyn liitetyistä componenteista. Jokaisella component-tyypillä on yksilöivä kokonaislukutunniste, joka kertoo, mitä entityn Signature-tietueen bittiä component-tyyppi vastaa. Entityt ja Signature-tietueet säilytetään omissa taulukoissaan niin, että entity ja sen Signature ovat taulukoissaan samassa indeksissä. Kuvassa 5 esimerkissä näytetään, kuinka tätä voidaan käyttää hyväksi.

```
1
2   #ifndef ENTITYID_HPP
3   #define ENTITYID_HPP
4
5   #include <cstdint>
6
7   typedef std::uint32_t EntityId;
8
9   #endif
10
```

KUVA 4. Entityn toteutus

Entityjen käsittelyyn määritellään joukko apufunktioita. Apufunktiot ovat entityn luomis- ja tuhoamisfunktiot, Signaturen hakufunktio entityn perusteella ja funktio, joka tarkastaa, onko entityllä tyyppiparametriksi annettu component.

```
48
49  // std::vector<EntityId> entities;
50  // std::vector<Signature> signatures;
51
52  Signature& signature(EntityId id)
53  {
54      int index = -1;
55      for(std::size_t i = 0; i < entities.size(); i++)
56      {
57          if(entities[i] == id)
58          {
59              index = i;
60              break;
61          }
62      }
63      return signatures[index];
64  }
65
```

KUVA 5. Esimerkki entity- ja Signature-taulukkojen käytöstä entityn käsittelyn apufunktiossa

Componentit toteutetaan esittelypelissä struct-rakenteella (kuva 6). Jokaisella component-tyypillä on yksi staattinen vakio, jonka arvo on uniikki ja joka kertoo, mikä entityn Signature-bittitaulun bitti kertoo, onko entityllä tämäntyyppinen component. Component-tyypillä on lisäksi tarvittava määrä alkeistyyppejä.

```cpp

#ifndef POSITIONCOMPONENT_HPP
#define POSITIONCOMPONENT_HPP

#include <cstddef>

struct PositionComponent
{
    static const std::size_t index;
    float x;
    float y;
};


#endif

```

KUVA 6. Esimerkki component-tyypistä

Componentien käsittelyyn määritetään joukko apufunktioita, kuten entitytyille. Componentien apufunktiot ovat componentin luonti- ja tuhoamisfunktiot ja funktio, joka palauttaa bittitaulun, jossa component-tyypin uniikin vakion määräämä bitti on asetettu ykköseksi.

Componentit säilytetään C++-kielen STL-kirjaston vector-tietorakenteessa jokainen component-tyyppi omassa tietorakenteessaan. Vector-tietorakenne valittiin, koska se säilyttää elementit muistissa yhtenäisesti (21). Entityyn liitetty component tallennetaan tyyppinsä vector-säiliöön käyttämällä entityn tunnistelukua indeksinä. Esittelypelissä ei käytetä edistyneempi tietorakenteita componentien säilytykseen, koska se on suhteellisen vaatimaton tehovaatimuksiltaan ja edistyneemmät tietorakenteet lisäisivät ohjelman monimutkaisuutta ilman merkittäviä hyötyjä.

Systemit toteutetaan esittelypelissä funktioilla. Funktiossa ensimmäiseksi käydään läpi kaikki entityt ja näiden Signaturet, joiden avulla luodaan lista entityistä, joilla on systemin toimintaan vaadittavat componentit (kuva 7). Lista iteroidaan läpi, ja componenteihin tehdään muutoksia systemin logiikan mukaan. Ohjelman pääsilmukassa määritellään järjestys, jossa systemit käydään läpi (kuva 8).

```cpp
std::vector<EntityId> movementAnimations;
std::vector<EntityId> spriteAnimations;
for(std::size_t i = 0; i < entities.size(); i++)
{
    if(Entity::has<MovementAnimationComponent>(entities[i]))
    {
        movementAnimations.push_back(entities[i]);
    }
    if(Entity::has<SpriteAnimationComponent>(entities[i]))
    {
        spriteAnimations.push_back(entities[i]);
    }
}
```

KUVA 7. Animaatio-entityt haetaan AnimationSystemin alussa

```cpp
if(components.turn.second.newTurn)
{
    TurnSystem(Entity::entities, components, data);
}

if(Entity::has<PlayerComponent>(components.turn.second.whosTurn))
{
    InputSystem(Entity::entities, components, data);
}
else
{
    AISystem(Entity::entities, components, data);
}

CollisionSystem(Entity::entities, components, data);
MovementSystem(Entity::entities, components, data);
CameraSystem(Entity::entities, components, data);
AnimationSystem(Entity::entities, components, data, deltaTime.asSeconds());

for(std::size_t i = 0; i < Entity::entities.size(); i++)
{
    Component::destroy<MovementComponent>(Entity::entities[i]);
}

RenderSystem(Entity::entities, components, data, window);
```

KUVA 8. Systemien keskinäinen järjestys ohjelmassa

4.4 Kokemukset esittelyohjelman toteutuksesta ECS-arkkitehtuurilla

Esittelypeli valmistui täyttäen kaikki esitetyt vaatimukset. Esittelyohjelmaa tehdessä todennettiin ECS-arkkitehtuurilla Hyödyt-kappaleessa esitettyjä väitteitä arkkitehtuurin eduista. Kuvassa 9 kuvakaappaus on valmistuneesta esittelypelistä.

KUVA 9. Kuvakaappaus esittelypelistä

Vuoronlaskussa oli virhe, joka aiheutti erään hahmon saavan ylimääräisiä vuoroja. Virhettä selvitettiin ajamalla koodia virheenjäljittimessä ja tarkastamalla vuoronlaskentaan käytettävien componentien arvoja eri systemien suoritusten aikana. Tällä tekniikalla paikansin ohjelmointivirheen nopeasti vuoronlaskennan suorittavassa systemissä. Tekniikka oli tehokas, koska ECS-arkkitehtuuri ohjaa ohjelmoijan kirjoittamaan suoraviivaista koodia, jossa vältetään ehtolausepuita ja logiikan murentamista useisiin funktioihin.

Törmäyksen tunnistamista tehtäessä piti refaktoroida koodia. Aluksi nuolinäppäimen painallukseen reagoitiin muuttamalla pelaajan hahmon sijaintia nuolen suuntaan, mutta koodia piti muuttaa niin, että pelaajan törmätessä johonkin sijaintia ei muuteta eikä liikkumisanimaatiota näytetä, vaan tehdään jotain muuta. Liikkumisen ja animoinnin toiminnot siirrettiin poissa käyttäjäsyötettä käsittelevästä InputSystemistä ja näiden sijaan se muokkaakin uutta MovementComponentia, joka säilyttää tiedot liikkumiseen liittyen. Tämän jälkeen suoritetaan törmäyksen tarkistamisesta vastaava CollisionSystem, ja sitten aikaisemmin InputSystemissä olleet sijainnin muuttamisen ja liikkumisen animoinnin aloittamisen toiminnot suoritetaan uudessa MovementSystemissä. Tämä onnistui helposti ECS-arkkitehtuurilla, koska arkkitehtuurissa Systemit eivät ole riippuvaisia toisistaan, joten niiden sisällä voidaan tehdä isojakin muutoksia, ilman että ohjelma hajoaa.

Peliä ohjelmoitaessa testattiin kehityksessä olevia toiminnallisuuksia ajamalla niitä yksistään testidataa vasten ja yhdessä muiden toiminnallisuuksien kanssa osana peliohjelmaa. Toiminnallisuuden testaaminen yksistään onnistui vaivattomasti arkkitehtuurilla, sillä jokainen toiminnallisuus meni omaan systemiin. Koska systemit eivät ole riippuvaisia toisistaan, voitiin testaaminen suorittaa alustamalla testidata, ajamalla systemin funktio ja tarkastamalla, vastaako testidatan muutos odotuksia. Kaikissa arkkitehtuureissa ohjelman osasen testaaminen ei onnistu näin yksinkertaisesti. Esimerkiksi oliopohjaisissa ohjelmistokehyksissä olioilla voi olla useita ajonaikaisia riippuvuuksia muihin olioihin, joita pitää testaamisen aikana jäljitellä sijaisolioilla. Ohjelmaa tai usean systemin kokonaisuutta voitiin testata ajamalla useampi system-funktio läpi ja tarkastamalla testidata tämän jälkeen. Koska toiminnallisuudet on erotettu toisistaan eri systemeihin, voitiin testattaessa tehdä helposti välitarkastuksia eri system-funktiokutsujen välissä koskematta funktioiden sisältöön.

5 YHTEENVETO

Opinnäytetyössä esiteltiin entity-component-system-ohjelmistoarkkitehtuuri. Arkkitehtuurin osat ja niiden väliset suhteet ja vuorovaikutukset määriteltiin ohjelmistoalan veteraanien kokemusten perusteella. Arkkitehtuurin vahvuuksiksi tunnistettiin koodin selkeys ja joustavuus sekä tuotettavan ohjelman testattavuus ja tehokkuus. Työssä myös kerrottiin, kuinka ECS-arkkitehtuuri voidaan toteuttaa ohjelmassa.

ECS-arkkitehtuuria käyttäen suunniteltiin ja tehtiin esittelyohjelma, jonka etenemisen perusteella arvioitiin arkkitehtuurin esitettyjä hyötyjä. Esittelyohjelma valmistui suunnitelman mukaan täyttäen kaikki vaatimukset. Ohjelman tekemisen aikana voitiin todeta, että arkkitehtuurin esitetyt hyödyt pitivät paikkansa. Ohjelmakoodi oli selkeää, mikä teki koodista helppolukuista sekä suoraviivaista seurata ja joustavaa, mikä näkyi muokatessa ja refaktoroitaessa koodia. Datan ja toiminnallisuuksien erottaminen ja toisistaan riippumattomat systemit tekivät esittelyohjelman testaamisesta helppoa.

Mitä esittelyohjelmalla ei voitu kokeilla, oli arkkitehtuurin tehokkuus. Opinnäytetyön aikarajoituksista johtuen ohjelmasta ei voitu tehdä niin vaativaa, että ohjelman tehokuuden vertailu olisi ollut mielekästä eikä useiden esittelyohjelmaversioiden tekemiseen vaihtoehtoisia arkkitehtuureja käyttäen olisi ollut aikaa. Opinnäytetyötä voisi laajentaa kehittämällä koejärjestely, jolla voitaisiin testata, kuinka ECS-arkkitehtuurin tehokkuus vertautuu muunlaisiin arkkitehtuureihin. Opinnäytetyön aikataulusta johtuen esittelyohjelman vaatimukset olivat myös suhteellisen yksinkertaiset eivätkä vastaa valmiin ohjelman ominaisuuksia ja laatua. Mahdollinen jatkotyö voisi olla laajempi ohjelma, jonka vaatimukset vastaavat paremmin valmiin ohjelman ominaisuuksia, ja näin ollen arkkitehtuurin sopivuutta ohjelmistokehitykseen voisi tarkastella todenmukaisemmassa tilanteessa.

LÄHTEET

1. Buttfield-Addison, Paris – Geldard, Mars – Nugent, Tim 2019. Konferenssiluento, Entity component systems and you: They're not just for game developers. New York, Yhdysvallat. Trianon Ballroom, O'Reilly Software Architecture Conference 6.2.2019. Saatavissa: https://www.youtube.com/watch?v=SFKR5rZBu-8. Hakupäivä 18.5.2021.

2. Martin, Adam 2007. T-machine.org. Entity Systems are the future of MMOG development – Part 2. Saatavissa: http://t-machine.org/index.php/2007/11/11/entity-systems-are-the-future-of-mmog-development-part-2/. Hakupäivä 1.11.2019.

3. Martin, Adam 2007. T-machine.org. Entity Systems are the future of MMOG development – Part 3. Saatavissa: http://t-machine.org/index.php/2007/12/22/entity-systems-are-the-future-of-mmog-development-part-3/. Hakupäivä 1.11.2019.

4. Zaks, Maxim 2018. Konferenssiluento, Entity Component System - A Different Approach to Game / Application Development. Nieuwegein, Alankomaat. Ordina HQ, Codestar Night 26.9.2018. Saatavissa: https://www.youtube.com/watch?v=lt4eL4RSx7k. Hakupäivä 18.5.2021.

5. Bilas, Scott 2002. Konferenssiluento, A Data-Driven Game Object System. Game Developers Conference 2002. Saatavissa: https://www.gdcvault.com/play/1022543/A-Data-Driven-Object. Hakupäivä 18.5.2021.

6. Martin, Adam 2007. T-machine.org. Entity Systems are the future of MMOG development – Part 1. Saatavissa: http://t-machine.org/index.php/2007/09/03/entity-systems-are-the-future-of-mmog-development-part-1/. Hakupäivä 1.11.2019.

7. Martin, Adam 2013. T-machine.org. Designing Bomberman with an Entity System: Which Components? Saatavissa: http://t-machine.org/index.php/2013/05/30/designing-bomberman-with-an-entity-system-which-components/. Hakupäivä 22.5.2020.

8. Ford, Timothy 2019. Konferenssiluento, Overwatch Gameplay Architecture and Netcode. San Francisco, Yhdysvallat. Moscone Convention Center, Game Developers Conference 27.2.2017-3.3.2017. Saatavissa: https://www.gdcvault.com/play/1024001/-Overwatch-Gameplay-Architecture-and. Hakupäivä 18.5.2021.

9. Martin, Adam 2014. T-machine.org. Data Structures for Entity Systems: Contiguous memory. Saatavissa: http://t-machine.org/index.php/2014/03/08/data-structures-for-entity-systems-contiguous-memory/. Hakupäivä 7.2.2020.

10. Nikolov, Stoyan 2018. Konferenssiluento, OOP Is Dead, Long Live Data-oriented Design. Bellevue, Yhdysvallat. Breckenridge Hall, CppCon 27.9.2018. Saatavissa: https://www.youtube.com/watch?v=yy8jQgmhbAU. Hakupäivä 18.5.2021.

11. Romeo, Vittorio 2016. Konferenssiluento, Implementation of a multithreaded compile-time ECS in C++14. Berlin, Saksa. Meeting C++ 2016 18.11.2016. Saatavissa: https://www.youtube.com/watch?v=51qSGUtaJwc. Hakupäivä 18.5.2021.

12. McMillan, Scott 2018. Konferenssiluento, Designing for Efficient Cache Usage. Sydney, Australia. Northside Conference Centre, Pacific++ 19.10.2018. Saatavissa: https://www.youtube.com/watch?v=3-ityWN-FdE. Hakupäivä 18.5.2021.

13. Martin, Adam 2015. T-machine.org. Entity ID’s: how big, using UUIDs or not, why, etc? Saatavissa: http://t-machine.org/index.php/2015/06/09/entity-ids-how-big-using-uuids-or-not-why-etc/. Hakupäivä 25.5.2020.

14. Martin, Adam 2009. T-machine.org. Entity Systems are the future of MMOG development – Part 5. Saatavissa: http://t-machine.org/index.php/2009/10/26/entity-systems-are-the-future-of-mmos-part-5/. Hakupäivä 1.11.2019.

15. Stein, Tobias 2019. Gamasutra. Entity Component System. Saatavissa: https://www.gamasutra.com/blogs/TobiasStein/20171122/310172/The_EntityComponentSystem__An_awesome_gamedesign_pattern_in_C_Part_1.php. Hakupäivä 2.11.2019.

16. Tietokannan normalisointi. 2020. Wikipedia. Saatavissa: https://fi.wikipedia.org/wiki/Tietokannan_normalisointi. Hakupäivä 31.5.2020.

17. Müller, Jonathan 2018. Konferenssiluento, Writing Cache Friendly C++. Berlin, Saksa. Vienna House Andels Hotel, Meeting C++ 2018 16.11.2018. Saatavissa: https://www.youtube.com/watch?v=Nz9SiF0QVKY. Hakupäivä 18.5.2021.

18. Linear Search. 2020. Wikipedia. Saatavissa: https://en.wikipedia.org/wiki/Linear_search. Hakupäivä 29.5.2020.

19. C++. 2020. Wikipedia. Saatavissa: https://en.wikipedia.org/wiki/C++. Hakupäivä 5.6.2020.

20. SFML. Saatavissa: https://www.sfml-dev.org/. Hakupäivä 5.6.2020.

21. std::vector. 2020. cplusplus.com. Saatavissa: https://www.cplusplus.com/reference/vector/vector/. Hakupäivä 5.6.2020.