Samuel Brooke

Moninpelitoteutukset Unreal 4 -pelimoottorilla

Metropolia Ammattikorkeakoulu

Insinööri (AMK)

Pelisovellukset

Insinöörityö

10.5.2018

Metropolia

Tekijä Otsikko Sivumäärä Aika	Samuel Brooke Moninpelitoteutukset Unreal 4 -pelimoottorilla 65 sivua 10.5.2018
Tutkinto	insinööri (AMK)
Tutkinto-ohjelma	Tieto- ja viestintätekniikka
Ammatillinen pääaine	Pelisovellukset
Ohjaajat	Miikka Mäki-Uuro, lehtori

Tämä insinöörityö käsittelee moninpeliin liittyviä haasteita ja niiden ratkaisutapoja sekä toteutuksia Unreal 4 -pelimoottorilla. Suurimpina haasteina ovat muun muassa pelaajien ja pelipalvelimen väliset latenssit. Menetelmät, joilla pyritään ratkaisemaan haasteiden aiheuttamia ongelmia liittyvät pelaajien sijainnin ennustamiseen ja ekstrapolaation sekä interpolaatiolla viivekompensointiin. Insinöörityön tavoitteena on luoda kattava yleysymmärrys sekä syventävä katselmus moninpelitoteutuksiin Unreal 4 -pelimoottorilla ja sen eri mahdollisuuksista moninpelitoteutuksissa.

Aluksi insinöörityö alustaa pohjan moninpelitoteutus esimerkeille. Unreal 4 -pelimoottorin perusteita käydään monipuolisesti läpi, tavoitteena saada yleisymmärrys moninpelimenetelmistä Unreal 4 -pelimoottorin ympäristössä. Pelimoottorin perusteisiin kuuluu muun muassa blueprint-skriptauskielen ja C++-kielen käytön opettelua.

Perusteiden jälkeen työssä syvennytään Unreal 4 -moninpeli perusteisiin, jotka ovat välttämättömiä moninpelitoteutuksien luomiseen pelimoottorissa. Käsiteltäviin aiheisiin kuuluu esimerkiksi replikointi- ja ”RPC”-funktioiden luontiin C++-kielellä ja blueprint-skriptauksella.

Unreal 4 -moninpeliperusteiden syvennyksen jälkeen, työssä käydään läpi käytännön moninpelitoteutus esimerkkejä, jotka on toteutettu Unreal 4 -pelimoottorilla. Esimerkit koostuvat kolmesta eri projekteista mitä tutkinut -ja kehittänyt pelimoottorilla, joista yksi on suuremman skaalan peliprojekti. Toteutukset käyttävät 4.18.3-versiota pelimoottorista.

Lopuksi työssä käsitellään omien kokemusten opit sekä luodaan yhteenveto Unreal 4 -pelimoottorin moninpelitoteutus mahdollisuuksista ja kuinka sitä voidaan käyttää erilaisiin toteutuksiin.

Avainsanat	Moninpelitoteutus, Unreal 4 -pelimoottori, replikaatio.

Author Title Number of Pages Date	Samuel Brooke Multiplayer Implementations in Unreal Engine 4 65 pages 10 May 2018
Degree	Bachelor of Engineering
Degree Programme	Information technology
Professional Major	game programming and applications
Instructors	Miika Mäki-Uuro, IT Lecturer

This thesis will go through general multiplayer game issues, the solutions that are related to them and multiplayer implementation examples created with Unreal Engine 4. The greatest difficulties related to multiplayer games are subjects such as latency between clients and game servers. These difficulties are generally attempted to be alleviated by using techniques known as client-side prediction, extrapolation and interpolation that are used to implement latency compensation. The goal of this thesis is to create a broad consensus of multiplayer solutions and implementations in Unreal Engine 4.

The thesis will begin with explaining a brief history of multiplayer games and the basics of networking related topics. This will be followed by a short crash course of how Unreal Engine 4 works with blueprint scripting and Unreal C++ language. After which it will go through how to implement basic functionality to understand multiplayer networking in the framework of the game engine. All of the examples are implemented with the 4.18.3 version of the game engine.

After the basics themes have been covered, the thesis will go through three different project examples that I've done during my time when using the game engine; one of which is a bigger multiplayer game project. Towards the and the thesis will provide a conclusion about how the game engine can be used to implement multiplayer games; also providing some insight and reflection of my own mistakes and things that I've learned while using Unreal Engine 4.

Keywords	Multiplayer implementations, Unreal Engine 4, replication.

Sisällys

1 Johdanto

Viimeiset puolitoistavuotta olen tutustunut aktiivisesti Unreal 4 -pelimoottorin käyttöön. Vuonna 2016 syksyllä, kolmannen lukuvuoden alussa meillä alkoi *Moninpeli*-kurssi. Kurssi oli tarkoitus toteuttaa yhteistyössä Metropolian, Amiedun ja Stadi-AV:n kesken. Päädyin itse projektiin, jossa tarkoituksena oli toteuttaa kaksintaistelu miekkailupeliä Unreal 4 -pelimoottorilla. Näin alkoi oma tutustuminen kyseiseen pelimoottoriin. Reilu puolitoista vuotta on kulunut siitä hetkestä. Kehitimme tammikuun 2018 asti kyseistä peliä Helsingin kaupungin rahoittamassa Digitalents-hankkeessa. Tavoitteena olisi saada peli julkaistua Steam-alustalle. Pakollisten kouluasioiden myötä päädyimme laittamaan hetkeksi projekti jäähyille, mutta toivomme kuitenkin, että saisimme julkaistua pelin ilmaiseksi versioksi 2018 kesän aikana.

Päädyin valitsemaan tämän työn aiheeksi moninpelitoteutukset Unreal 4 -pelimoottorilla, koska minulla oli aikaisemmasta tehdystä menetelmäopintotyöstä ”Unreal Engine 4: Moninpelitoteutukset blueprint-ympäristössä” oleva aihe mitä halusin vielä laajentaa. Edellisessä työssä käsittelin melko ympäripyöreästi tiettyjä aiheita ja käsittelin vain blueprint skriptaus menetelmiä sekä toteutuksia aiheeseen liittyen. Huomasin, miten vähäistä tietoa oli saatavilla suomenkielellä Unreal 4 -pelimoottorista, joten otin tässä työssä tavoitteissani huomioon sen, että tästä työstä voisi olla hyötyä muille Unreal 4 -pelimoottorista ja moninpelitoteutuksista kiinnostuneille, vaikka kuitenkin kyseessä on tutkielmatyö. Lisäksi halusin laajentaa aihetta, jotta se ei ainoastaan käsittelisi Unreal 4 -pelimoottorin moninpelitoteutuksia. Tavoitteena oli selvittää yleisiä ongelmia ja ratkaisutapoja, jotka liittyivät moninpelien kehittämiseen, niitä olivat esimerkiksi viive ja sen kompensoiminen moninpeleissä.

Tässä työssä aion ensin käydä läpi moninpeliin liittyviä yleisiä aiheita, niiden ongelmia ja ratkaisutapoja, mitkä olisivat syytä osata ja pitää mielessä, kun lähtee toteuttamaan moninpelattavaa peliä. Tämän jälkeen käyn läpi lyhyesti Unreal 4 -pelimoottorin perusteita, workflow’ta, blueprint-skriptauksta, C++ -kielen käyttöä ja niiden eroavaisuuksia. Sen jälkeen työssä syvennyn Unreal 4 -verkkoarkkitehtuuriin ja moottorin rakenteeseen. Näitä aiheita olisi syytä ymmärtää perustasolla ennen kuin lähtee toteuttamaan moninpelattavaa peliä tällä pelimoottorilla. Työn loppupuolella päästään itse moninpelitoteutus aiheisiin ja esimerkkeihin, joita käsittelen blueprint-skriptausta sekä C++ -kieltä eri

perspektiiveistä. Perusteiden jälkeen annan konkreettisia esimerkkejä molemmilla ohjelmointitavoilla ja esittelen myös oman ”Rising Cut” -nimisen pelin toteutuksia.

2 Yleiskuva moninpelimenetelmistä

Työn ensimmäisessä osiossa käydään lyhyesti läpi moninpelin historiaa, jonka jälkeen aihe siirtyy moninpelien yleisiin haasteisiin ja ratkaisutapoihin.

2.1 Moninpelien historia

Ensimmäisiä moninpelejä kehitettiin 1970-luvulta alkaen. Muutamaa poikkeusta lukuun ottamatta kyseiset pelit olivat kuitenkin vain paikallisesti moninpelattavia. Moninpelien suosio alkoi yleistyä vasta 1990-luvun puolessa välissä samanaikaisesti internetin käytön yleistymisen myötä. Kehittäjät huomasivat netin yleistymisen mahdollisuudet ja lähtivät sitten kehittämään pelejään hyödyntämällä uutta teknologia. PC-pelit kuten ID Softwaren ”Quake” (1996) ja Epic Gamesin ”Unreal” (1998) olivat edelläkävijöitä internetin välisessä moninpelien kehittämisessä.

Moninpelien kehitys ei rajoittunut vain fps-genreen. Myös ensimmäisiä MMO-pelejä (”Massively Multiplayer Online games”) alkoi ilmestymään. MMO:lla tarkoitetaan moninpeliä, joka tukee monta sataa tai tuhatta pelaajaa, jotka voivat pelata samanaikaisesti samaa peliä. Suurin osa MMO-peleistä olivat aluksi roolipeli muottiin pohjautuvia. Ensimmäisiä isoja merkillisiä pelejä olivat ”UltimaOnline” (1998),”EverQuest”(1999) ja Blizzard-peliyhtiön kehittämä ”World of Warcraft” (2004).

Moninpelien suosio siirtyi myös konsoleihin 2000-luvun alussa. Suurimpien konsolipeli yritysten välinen kilpailu jatkui omilla kehittämillään moninpelipalveluilla, kuten Sonyn ”Playstation Networkllä” ja Microsoftin ”Xbox-livellä”. Erityisen menestyviä moninpelejä kyseisille alustoille olivat muun muassa Xbox-konsolille kehitetty ”Halo” -niminen fps-pelisarja. Vaikka ”Halo” ei ollut ensimmäinen fps-peli konsolille, se kuitenkin asetti

tietynlaisen standardin fps-moninpeleille konsolimarkkinoilla. Tämän jälkeen useita fps-moninpelejä julkaistiin konsoleilla. Suosion seurauksena ”Activision”-nimisen peliyrityksen kehittämä fps-pelisarja ”Call of duty” sai tuulta purjeisiin, jonka jälkeen alkoi reilun kymmenen vuoden ”Call of duty” -suosio.

Älypuhelimien tulo avasi kokonaan uuden markkinan pelien kehittämiseen ja tätä myötä myös mobiilipelien kehittäjät alkoivat tuottamaan moninpelejä. Suurin osa mobiilimoninpeleistä oli strategia- ja manageerointipelejä, jotka olivat ”Asynchronous” eli eivät vaatineet oikea-aikaista datasiirtoa kuten tyypillisissä moninpeleissä. Kyseiset pelit ilmoittivat pelaajalle, milloin heidän vuoronsa oli, pelaajalla oli melkein loputtomasti aikaa suorittaa omavuoronsa. Pelit kuten Supercellin kehittämä ”Clash of Clans” hyödyntävät ”Asynchronous”-moninpelityyliä. Kyseinen muotti soveltui hyvin mobiilipeleillä, sillä verkkojenyhteys mobiililaitteissa alkoi parantua huomattavasti vasta 2010-luvusta eteenpäin. Tämän jälkeen oikeanajan moninpelejä alkoi yleistymään mobiililaitteille, kuten ”Blizzard”-nimisen peliyhtiön julkaisema korttipeli ”Hearthstone” (2014). (1.)

2.2 Moninpelitoteutuksen haasteet ja tekniikat

Moninpelien kehittäminen on yksi hankalimpia ohjelmointiin liittyvistä haasteista. Kyseinen aihe on niin laaja, että on vaikeata osata vaan tiettyjä osia aiheen kokonaisuudessa. Tämä työn tarkoitus on kuitenkin käsitellä Unreal 4 -pelimoottorin moninpelitoteutukset pääasiassa. Näin ollen käsittelen seuraavat haasteet mahdollisimman yleisellä tasolla, jotta tekstin pituus ei paisuisi liikaa. Pyrin kuitenkin kirjoittamaan tarpeeksi tietoa, että lukija saisi tarvittavan ymmärryksen hahmottamaan yleisiä haasteita, jotka liittyvät moninpelien kehittämiseen. Suuri osa matalamman tason haasteista ovat jo itse moottorissa toteutettuja, joten mainitsen niistä kokemani tarpeen mukaan.

2.2.1 Korkeantason arkkitehtuuriset haasteet

Yleinen ero korkean ja matalan tason haasteista voi kuvailla yleisesti seuraavalla tavalla. Matalalla tasolla käsitellään, miten bitit lähetetään palvelimen ja pelaajien välillä, kun taas korkealla tasolla mietitään, mitä bittejä kuuluisi edes lähettää. Toisin sanoen mietimme korkealla tasolla, miten pelaajat välittävät toisilleen tietoa ja millä tavalla. Matalalla tasolla mietitään itse tiedonsiirron tekniikkaa ja toteutusmenetelmää.

Kuvitellaan tilanne moninpelissä, jossa pelaajaa A ja B taistelevat vastakkain. Moninpelissä tulee aina vastaan tällaisia tilanteita, jossa joudumme tekemään päätöksen tapahtuman lopputuloksesta. Ei voi olla tilannetta, jossa molemmat pelaajat luulevat tappaneensa toisensa samanaikasesti.

Yleisimpiä arkkitehtuurimalleja joita käytetään moninpelien toteutuksissa, ovat ”Client-Server”- ja ”Peer-to-Peer”-moninpeliarkkitehtuurimallit. Molemmissa suunnittelumalleissa on omat hyödyt ja haittapuolensa. Molemmilla suunnittelumalleilla voidaan kuitenkin ratkaista yllä mainittua yleistä moninpelitilanteen ongelmaa. (2.)

2.2.2 ”Client-Server”-moninpeliarkkitehtuurimalli

”Client-server”-arkkitehtuurimallissa kaikki asiakkaat (”Client”) ottavat yhteyden pelipalvelimeen (”Server”). Palvelin on vastuussa, eli saa niin sanotun auktoriteetin ja tekee viimeisen päätöksen tärkeimmistä peliin liittyvistä tapahtumista. Pelaaja välittää itsestään tarvittavat tiedot palvelimelle. Palvelin tarkistaa, ovatko kyseiset liikkeet todellisia tai yhtenäisiä palvelimen tiedon kanssa. Palvelin joko hyväksyy tai hylkää pelaajan tekemät muutokset omista liikkeistä ja päivittää sitten pelaajan tiedot muille pelaajille.

Tässä mallissa saadaan yksi yhdenmukainen kuva koko pelin tilanteesta palvelimen avulla. Yksittäiset pelaajat eivät pysty helposti tahattomasti tai tahallisesti (huijaamalla) vaikuttamaan koko pelin tilanteeseen epäreilulla tavalla. Malli ei kuitenkaan ole täydellinen, vaan nopeasti voi huomata, että palvelimen laatu vaikuttaa koko pelikokemukseen. Palvelin tulee täysin riippuvaiseksi sen koneen tehokkuudesta ja yhteyden siirtonopeudesta, sillä sen sijaan, että pelaajat hoitaisivat osan lasku suorituksista itse, palvelin on nyt täysin vastuussa niiden suorittamisesta. (2.)

2.2.3 ”Peer-to-Peer”-moninpeliarkkitehtuurimalli

Kyseisessä mallissa suurin hyöty tulee, kun lasku suorituksia jaetaan pelaajien kesken. Tätä mallia kutsutaan nimellä ”Peer-to-Peer”. Suurin hyöty kuitenkin koituu myös kohtaloksi tälle mallille, sillä laskusuoritukset ovat suoraan verrannollisia päätöksentekoon pelissä. Toisin sanoen pelaajat voivat hyvin helposti huijata tässä arkkitehtuurimallissa. Esimerkiksi pelaajaa voi olla ottamatta huomioon tietopaketteja, jotka liittyvät häneen kohdistuvaan vahinkoon. Hänestä tulee näin ollen kuolematon.

Ongelmat eivät lopu siihen, vaan peli altistuu virheisiin, kun pelaajat voivat kaikki tehdä päätöksiä pelin tilanteesta. On yleistä, että pelissä tulee tilanne, missä pelin tiedot eivät ole yhteneväisiä kaikkien pelaajien koneilla. (2.)

2.2.4 Hybridimalli

Todellisuudessa suurin osa arkkitehtuurimalleista ovat hybridejä, eli ne pyrkivät ottamaan parhaimpia puolia molemmista malleista. Hybridimallit pohjautuvat ”Client-Server”-malliin, mutta ero on siinä, että annetaan pelaajalle mahdollisuus ja auktoriteetti tehdä päätöksiä omasta sijainnistaan. Tämä yhdistetään palvelimen tekemiin tarkistuksiin, jotta vähennettäisiin huijausmahdollisuuksia. Toisaalta aina kun annetaan pelaajalle mahdollisuus tehdä itsestään päätöksiä, huijaamisen mahdollisuudet nousevat. Myös *Unreal 4* hyödyntää tämän tyyppistä ”Client-Server”-mallia. (2.)

2.3 Tiedon välitys ja latenssin haasteet

Seuraavassa työn osiossa käydään läpi tiedon välityksen ja latenssin haasteiden eri aiheita ja loppupuolella käsitellään yleisesti käytettyjä ratkaisutapoja haasteiden vaikutusten vähentämiseen.

2.3.1 Latenssi

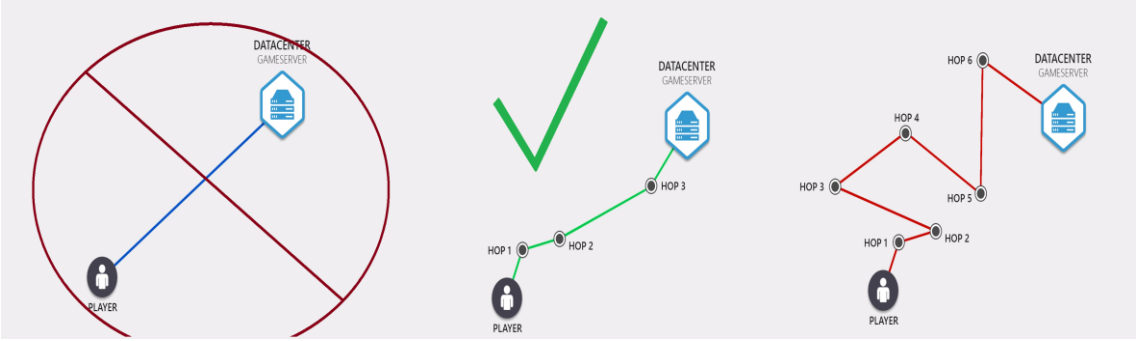
Verkkomoninpeleissä on ominaista pelaajan ja palvelimen tiedonvälityksen aiheuttama viive, eli latenssia. Usein yleisessä puhekielessä esiintyy sanoja kuten ”Lag” tai ”Ping” kun kuvaillaan kyseistä ilmiöitä. ”Lag”-sanalla kuvaillaan yleensä itse viiveen tunnetta pelaajan tekemään liikkeeseen suhteessa mitä palvelin välittää takaisin pelaajan ruutuun. ”Ping”-sanalla yleensä kuvataan viiveen ajan pituutta millisekunneissa, eli kauanko kestää, että pelaaja lähettää ensin tietoa palvelimelle ja saa palvelimelta päivityksiä muista pelaajista tai pelin tilasta takaisin. Todellisuudessa ”Ping”-sanalla tarkoitetaan vain yleistermiä, jolla mitataan ”RoundTripTime”(RTT), eli aikaa, jossa tietopaketti pääsee kohteesta takaisin lähettäjälle. Korkea ”Ping” aiheuttaa siis ”Lag”-viiveilmiötä, joka joissain peleissä kuten fps-peleissä aiheuttaa epäoikeudenmukaisia etuuksia muita pelaajia kohtaan. Joissakin peleissä tämä otetaan huomioon niin, että jos pelaajalla on liian korkea viive, hänen yhteys katkaistaan palvelimelta. Tämä ratkaisu ei tietysti ole aina

paras vaihtoehto, sillä se rajoittaa ihmisiä, jotka asuvat kaukana toisistaan pelaamasta keskenään. (4; 5.)

2.3.2 Yhteysreitit palvelimelle, päivitysnopeudet ja simulaationopeus

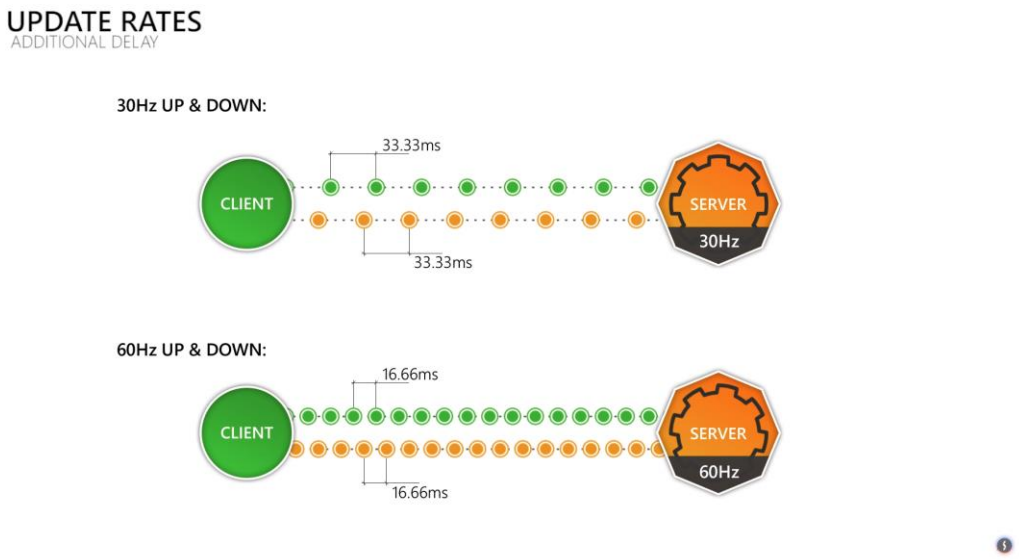
Latenssi kulkee käsikädessä moninpelien kanssa. Rajana on valon nopeus, sillä emme fysiikanlakien mukaan pysty kuljettamaan dataa valon nopeutta nopeammin. Etäisyyden lisäksi huomioon otettavia seikkoja verkkoviiveissä ovat, miten pelaajat pääsevät yhdistymään pelipalvelimelle ja itse palvelimen päivitysnopeus eli ”update-rate”.

Viiveeseen liittyy yhteysreitti joka pelaajan koneelle laaditaan. Se saattaa kulkea monen eri reitittimen läpi. Parhaassa tapauksessa pelaajan reititin yrittää käyttää lyhintä mahdollista reittiä. Todellisuudessa voi käydä poikkeustilanteita, joissa paras mahdollinen reitti ei toimi, reititin joutuu valitsemaan pidemmän tai huonomman laatuisen reitin aiheuttaen lisää matkaa ja viivettä. Jos reitti on huono, lisääntyneen viiveen lisäksi pelaaja voi kokea datanmenetystä eli ”packetloss”. Tällä tarkoitetaan sitä, kun tietty lähetetty tietopaketti epäonnistuu pääsemästä pelaajalle perille. Seuraavassa kuvaesimerkissä käy ilmi, miten asiakkaan yhteysreitti määritellään pelipalvelimelle.



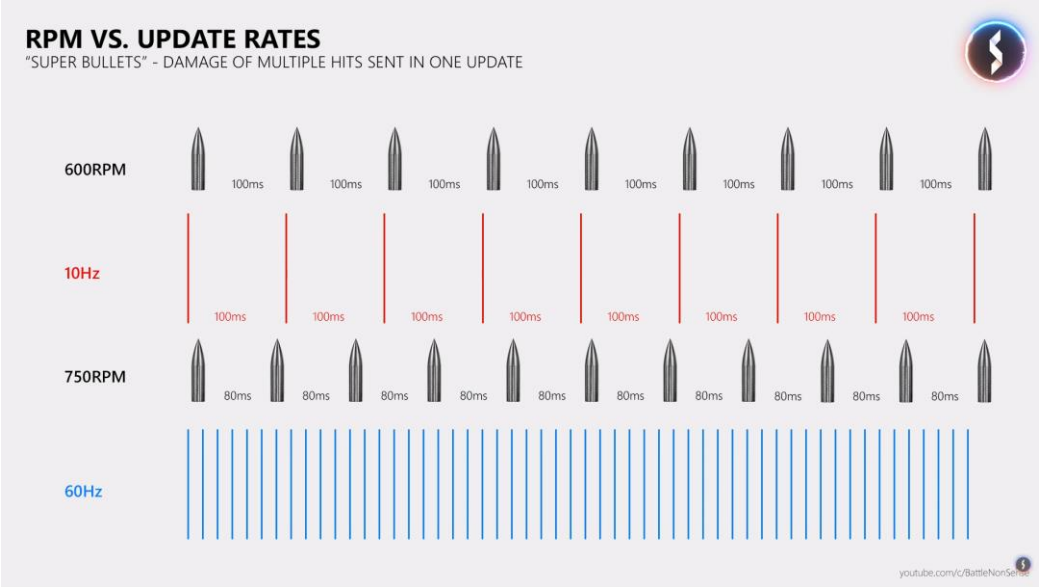
Kuva 1. Kuvaesimerkki reitittimen yhteydestä pelipalvelimelle. Vasemmalla on vääristynyt tapa mitata etäisyyttä. Keskellä näkyy, miltä reitittimen matka palvelimelle oikeasti näyttää. Oikealla näkyy poikkeustilanne, jossa reititin valitsee huonon reitin. ”Battlenonsense”-Youtube-kanavan videosta. (5.)

Tämän lisäksi viiveeseen vaikuttaa pelaajan ja palvelimen välinen päivitysnopeus, eli kuinka tiheällä tahdilla pelaaja pystyy lähettämään ja vastaanottamaan tietoa palvelimelta. Seuraava kuva havainnollistaa, miten päivitysnopeus toimii.



Kuva 2. Kuvaesimerkki päivitysnopeuksista, kuva otettu ”Battlenonsense”-Youtube-kanavan videosta. (5.)

Kuvassa 2 näkyy, miten päivitysnopeudet vaikuttavat viiveeseen. Korkeammalla päivitysnopeudella voidaan vähentää aikaa päivitysten välillä, jotka aiheuttavat lisää viivettä. Hitaat päivitysnopeudet voivat aiheuttaa muun muassa fps-peleissä ylimääräisen viiveen lisäksi niin sanotun ”superbullet”- eli ”superluoti”-ilmiön, joka käytännössä tarkoittaa, että pelaaja ottaisi monen luodin verran vahinkoa yhden päivityksen aikana. Seuraava kuva havainnollistaa ”superluoti”-ilmiötä.



Kuva 3. Kuvaesimerkki esittää ”superluoti”-ilmiötä. Kuva otettu ”Battlenonsense”-Youtube-kanavan videosta. (5.)

Kuvassa 3 voimme nähdä miten ”superluoti”-ilmiö toimii. Oletetaan että palvelimen päivitysnopeus on vain 10 Hz, eli jokaisen päivityksen välissä on 100 ms tyhjää aikaa, ennen kuin seuraava päivitys tulee. Tässä esimerkissä ja useimmissa ampumispeleissä käytetään aseita, joissa on korkeita tulinopeuksia, 600 luotia minuutissa(rpm) tai korkeampia. 600 rpm:n ase ampuu uuden luodin aseesta 100 ms välein. Eli jos ase ampuu nopeammalla tahdilla, se ampuu uuden luodin nopeammin kuin palvelimen päivitysnopeus. Tämä tarkoittaa, että palvelin päivittää kahden luodin verran vahinkoa yhden päivityksen aikana, joka aiheuttaa tilanteen, jossa vastaanottava pelaaja kuvittelee saaneensa tupla-osuman yhdestä luodista. Päivitysnopeuteen vaikuttaa suoraan pelinpalvelimen kykyä tuottaa tietoa ja kuinka tiheästi se pystyy tuottamaan sitä yhden päivityksen aikana, eli niin sanottu simulaationopeus (”tickrate”). Kuva 4 esittää esimerkin simulaationopeuedesta.

Kuva 4. Kuvaesimerkki miten ”tickrate” toimii. Kuva otettu ”Battlenonsense”-Youtube-kanavan videosta. (5.)

Kuvan 4 esimerkissä on palvelimella 60 hertsin simulaationopeus, joka mahdollistaa 60 hertsin päivitysnopeudet pelaajan ja palvelimen välillä. Päivitysnopeuden tiheys ei kuitenkaan ole ainoa tekijä, vaan on yhtä tärkeätä, että palvelin saattaa päivityksen loppuu mahdollisimman nopeasti tai edes kyseisen simulaatiopäivityksen aikana.

Kuva 5. Esimerkki simulaatioaskeleen vaiheesta, simulaatiopäivityksen aikana. Kuva otettu ”Battlenonsense”-Youtube-kanavan videosta. (5.)

Kuvan 5 esimerkissä voidaan havainnoida paremmin, miten simulaatioaskel toimii simulaatiopäivityksen aikana. Palvelin ottaa vastaan saadun tiedon pelaajista ja maailmasta, prosessoi sen ja sitten simuloi tuloksen, jonka jälkeen palvelin odottaa seuraavaa simulaatiopäivitystä. Mitä nopeammin palvelin kykenee suorittamaan loppuun kyseisen simulaatioaskeleen, sitä paremmalta pelikokemus kuten esimerkiksi osumatarkkuus tuntuu pelaajille. Jos palvelin epäonnistuu käsittelemään simulaatioaskeleen tiedon simulaatiopäivityksen aikana, pelissä se voi ilmentyä monessa eri muodossa, kuten muiden pelaajien ”kuminauhailua” paikasta toiseen. Ammutut luodit eivät rekisteröidy yms. (4; 5.)

2.4 Yleisesti käytettyjä ratkaisutapoja moninpelihaasteissa

Osiossa käydään moninpeliin liittyviä ratkaisutapoja, joilla yritetään parantaa moninpelin pelikokemusta. Aiheita ovat muun muassa ”vakiintuneet” pelipalvelimet, pelaajan sijainnin ennustus ja viiveen kompensoiminen.

2.4.1 ”Vakiintuneet”-pelipalvelimet

”Client-Server”-malleissa useimmiten käytetään niin sanottuja ”vakiintuneita pelipalvelimia” (”dedicated servers”), johon pelaajat voivat kaikki yhdistyä. Tämä yleensä takaa sen, että peli pyörii korkeatasoisella laitteistolla ja internetyhteydellä. Useimmat moninpelit yrittävät myös yhdistää pelaajia mahdollisimman lähellä oleviin palvelimiin, jotta latenssi olisi mahdollisimman pieni kaikille yhdistäville asiakkaille.

2.4.2 Pelaajan sijainnin ennustus

Latenssi on asia, jota ei voida saada pois kokonaan moninpeleistä. ”Vakiintuneiden” pelipalvelimien käyttö ei myöskään yksistään riitä hyvän pelikokemuksen takaamiseksi. Suurin osa nykyajan moninpeleistä hyödyntävät niin sanottua pelaajan sijainnin ennustamista (”Client side prediction”), jolla koitetaan ennustaa etukäteen pelaajan sijaintia ja liikkeitä. Pelaajalle annetaan valtuus omista liikkeistään, mutta taustalla on kuitenkin pelipalvelin, jolla on auktoriteetti tehdä lopullinen päätös pelaajan sijainnista.

Sijainnin ennustamista implementoidaan generoimalla käyttäjäkomentoja pelaajan liikkeen sisääntuloista. Tämä tallennetaan pelaajan puolella ja sitten komentotiedot lähetetään palvelimelle. Pelaajan koneella tallennetaan myös aika, milloin tiedot on tallennettu, sillä itse ennustus algoritmi hyödyntää näitä tallennettuja käyttäjäkomentoja ja niiden aikaa, milloin ne on tallennettu. Seuraavassa koodi esimerkissä käy ilmi, miltä se näyttäisi.

```
typedef struct usercmd_s
{
        // Interpolation time on client
        short           lerp_msec;
        // Duration in ms of command
        byte            msec;
        // Command view angles.
        vec3_t  viewangles;
        // intended velocities
        // Forward velocity.
        float           forwardmove;
        // Sideways velocity.
        float           sidemove;
        // Upward velocity.
        float           upmove;
        // Attack buttons
        unsigned short buttons;
```

```
        //
        // Additional fields omitted...
        //
} usercmd_t;
```

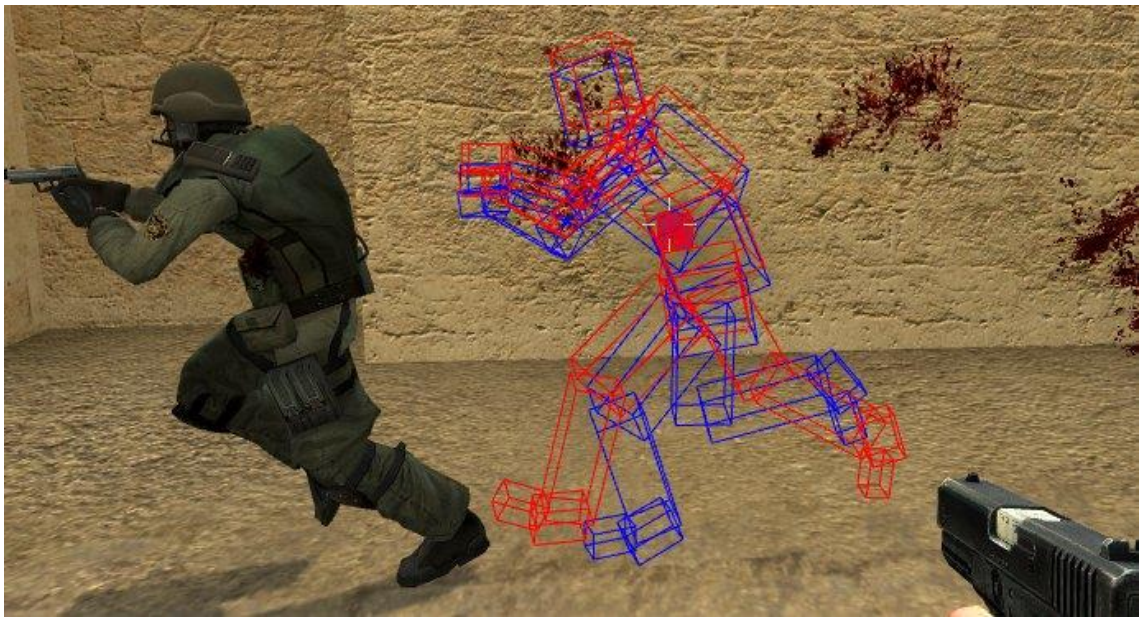
Esimerkkikoodi 1. Esimerkkikoodi kuvastaa miten ja millaista tietoa voidaan tallentaa pelaajan sijainnin ennustamista varten.

Esimerkkikoodissa 1 näkyy, miten ”Valve”-yhtiö on toteuttanut käyttäjän sisääntulon tallentamisen omilla pelimoottoreilla. Tärkeimmät muuttujat ovat esillä pätkässä, esimerkiksi ”Viewangles”-muuttuja kuvastaa pelaajan katsomis suuntaa, joka on hyvin tärkeä tieto välittää muille pelaajille. ”Msec”-muuttuja kuvastaa kyseisen liikkeen tallennusaikaa. Pelaajan eri suunnille on omat muuttujat, jotka kuvastavat pelaajan painamaa suuntaa. Tämä on vasta ensimmäinen askel, joka antaa algoritmille oikeat muuttujat, joilla se voi tehdä tarvittavan työn. Jos pelaajalla olisi 500 ms:n latenssi tässä tilanteessa, hän kokisi toistaiseksi vielä muutoksen vasta 500 ms:n päässä omalla ruudullaan.

Itse ennustava algoritmi toimii niin, että se käyttää palvelimen viimeisen saadun tiedon liikkeestä alkupisteenä. Palvelimen viimeisin vahvistus osoittaa, mikä käyttäjäkomento oli viimeksi käytetty palvelimella. Jos pelaajan peli pyörii 50 ruutupäivityksellä (frames persecond) ja hänellä on 100 ms latenssi, hän on tallentanut 5 käyttäjäkomentopakettia palvelimen viimeksi vahvistetun paketin edellä. Nämä viisi komentopakettia simuloidaan pelaajan puolella käyttämällä vastaavan tyyppistä logiikkaa, mitä palvelin käyttää pelaajan sijainnin simuloinnissa. Lopputuloksena vältytään viiveen tunteesta oman pelaajan ja muiden pelaajan liikkeissä korkeammalla latenssilla. (6.)

2.4.3 Ekstrapolaatio ja interpolaatio-viivekompensaatio

Seuraava tärkeä asia liittyen moninpelien sulavaan moninpelattavuuteen on toteutustapoja, joilla piirretään toiset pelaajat asiakkaan puolella pelaajille. Yleisimpiä toteutusmenetelmiä, joita käytetään, ovat piirrettävien pelaajaobjektien ekstrapolaatio- ja interpolaatio. Ekstrapolaatio- ja interpolaatio ovat matemaattisia menetelmiä, joilla koitetaan ennustaa ja laskea uusia arvoja kahden mitatun arvon välille. Moninpelikontekstissa nämä uudet arvot ennustetaan ja lasketaan pelaajien sijaintien, viiveen ja nopeuden perusteella. Seuraavassa kuvaesimerkissä voidaan paremmin hahmottaa, miten Counter-Strike:Sourcessa toimii viivekompensaatio.



Kuva 6. Kuvaesimerkissä nähdään, miten ”Valve”-yhtiön kehittämä ”Counterstrike-source”-moninpelin viivekompensaatio toimii. Sinisenväriset osumalaatikot kuvastavat palvelimen puoleisia ”taaksepäinkelattuja” osumalaatikoita, kun taas punaiset ovat asiakkaan puolella. Kuvan tilanne ei liity suoraan alla olevaan tekstiin. (6.)

Ekstrapolaatiometodissa simuloidaan pelaajan tai objektin sijaintia tulevaisuuteen pelaajan viimeksi päivitetystä sijainnista. Otetaan esimerkiksi fps-moninpelitilanne, jossa on piirrettävä pelaaja, joka juoksee vaakasuoraan asiakkaan ruudussa 500 ups:n (”units per second”) vauhdilla ja hänellä on 100 ms:n viive palvelimeen. Asiakkaan puolella oletettaisiin piirrettävän pelaajan sijaintia olevan 50 ”unit”-mittaa (pelimaailman etäisyysmitta), sen viimeksi päivitetystä sijainnista. Näin ollen jos asiakkaan puoleinen pelaaja ampuisi piirrettävää pelaajaa kohti, niin sijainnin pitäisi olla oikea, riippumatta piirrettävän pelaajan viiveestä palvelimeen. Yleisiä ongelmia, jotka seuraavat tätä metodia, ovat piirrettävän pelaajan epäjohdonmukaiset liikkumiset ja sen niin sanottu ”kuminauhailu”. Toinen ongelma ekstrapolointimenetelmässä on esimerkiksi fps-moninpeleissä yleisimmin käytettyjen fysiikkamallien seuraamat epärealistiset voimat, joita pelaaja voi hyödyntää epäreilusti pelissä. Tästä aiheutuu virheellisiä ekstrapoloituja sijainteja. Kehittäjät voivat vähentää ekstrapolaatioaikaa, joka vähentää sen aiheuttamaa ”kuminauhailua”, mutta tämän myötä pelikokemus silti kärsii, jos pelaajilla on korkea viive ja pelaajat joutuvat ennakoimaan laukauksia joka tapauksessa.

Toinen metodi, jolla voidaan määrittää piirrettävän pelaajan sijaintia, kutsutaan interpolaatioksi. Interpolaatio moninpeleissä käytännössä tarkoitetaan objektin sijainnin siirtämistä lähimenneisyyteen pelaajan viimeksi päivitetyn sijainnin perusteella. Esimerkiksi

jos palvelin lähettää tarkalleen 10 päivitystä sekunnissa pelimaailman tilanteesta, peli asettaa 100 ms:n interpolointiin viiveen pelinpiirtämisessä. Kun peli piirretään, piirrettävän pelaajan sijaintia interpoloidaan viimeksi päivitetyn sijainnin ja yhden ruutupäivityksen edeltävästä sijainnin välillä. Toisin sanoen palvelimelta saadaan uusi sijainti joka 100 ms, ja pelaajalla on saman verran aikaa päivittää itsensä uuteen sijaintiin ennen seuraavan päivityksen saamista.

Tässä metodissa voidaan myös yhdistää ekstrapolaatio-tekniikkaa, jos palvelimen ja pelaajan välillä tippuu tietopaketteja aiheuttaen ”kuminauhailua”. Vaihtoehtoisesti voidaan määrittää, että pelaajan sijaintia päivitetään vasta seuraavalla saadulla päivityksellä, Tämän seurauksena pelaajan liikkuminen näyttäisi ”värisevän” seuraavaan sijaintiin. Kolmas ratkaisu saman esimerkin kontekstissa on lisätä interpolaation aikaa 200 ms, jossa tiputetaan yhden päivityksen tieto kokonaan ja näin ollen pelaajan sijainti interpoloidaan sitä seuraavan saadun sijainnin välillä. Tämä on tietysti kompromissi, sillä kyseinen piirretty pelaaja olisi näin ollen vaikeampi osua aseella.

Tiivistettynä näillä menetelmillä voidaan kompensoida viivettä pelaajien ja palvelimen välillä. Viiveen kompensaatiolla siis tarkoitetaan palvelimen ja pelaajien välisen liikkeiden ja komentojen viiveen normalisointia. Sen voi yleisemmin kuvailla ottamalla askeleen menneisyyteen pelipalvelimen puolella ja tarkastella pelimaailman tilaa samaan aikaan, kun pelaaja tekee itse liikeitä ja komentoja. Viivekompensaatio mahdollistaa jokaisen yhdistävän pelaajan saamaan suhteellisen viiveettömän pelikokemuksen riippumatta heidän viiveistään pelipalvelimelle. Gabriel Gambettan (lähdeluettelolinkki numero 24) toteuttamassa livedemossa voidaan hahmottaa tarkemmin, miten viivekompensaatio toimii. (6; 7; 8; 24.)

2.4.4 Viivekompensaation vaikutukset pelisuunnitteluun

Latenssin viivekompensaatiosta huolimatta, tulee epäjohdonmukaisuuksia, jotka vaikuttavat pelinsuunnitteluun. Ennen viivekompensaation esiintymistä moninpeleissä, ongelmia olivat esimerkiksi aikaisemmin mainittu fps-moninpeliin liittyvä tähtäämisennakointia. Pelaaja joutui oman ja vihollispelaajan viiveen perusteella aina ennakoimaan eri tavalla pelaajan liikkeitä. Tämä epäjohdonmukaisuus aiheuttaa epärealistisen tunteen tähtäämisessä ja tekee vihollispelaajan liikkeistä ennakoimattomia. Viivekompensaation myötä

kuitenkin tulee myös omia epäjohdonmukaisuuksia, jotka vaikuttavat pelinsuunniteluun. Yleisimmin näissä tapauksissa osumaa ottava pelaajaa on se joka huomaa näitä ongelmia. Tyypillinen tilanne on tunne, kun osumaa ottava pelaaja ottaa osumaa ”seinän takana”. Tämä tapahtuu esimerkiksi, kun korkeaviiveinen pelaaja ampuu matalaviiveistä pelaajaa, kun hän menee nurkan taakse. Syy tähän on, että korkeaviiveinen pelaaja näkee menneessä matalaviiveisen pelaajan sijainnin.

Kuva 7. Battlefield 1:n viivekompensaation ratkaisun testiesimerkki. Kuva on otettu ”Battlenonsense”-Youtube-kanavan tehdyistä kokeista. (5.)

Kuvassa 7 näkee miten tämä Battlefield 1 -pelin kehittäjät ”Dice” ovat yrittäneet ratkaista tätä ongelmaa. Tilanteessa on kaksi pelaaja, jossa pelaaja yksi juoksee huoneiston läpi suojaan. Kyseisellä pelaajalla 47 ms:n viive palvelimelle. Pelaaja kaksi yrittää ampua ohi juoksevaa pelaajaa ja hänellä on 147 ms:n viive palvelimelle. Ykköspelaajan asiakas ajaa osumarekisteröinti koodin saatuaan informaation osumasta toiselta pelaajalta. Tämän jälkeen palvelin ajaa saman tarkistuksen osumasta, jotta se voisi varmistaa, oliko osuma totta vai ei. Jos osuma pitää paikkaansa ykköspelaajalta vähennetään elopisteitä normaalisti. Tätä toimintoa kutsutaan ”Client-side authoritative” -tarkistukseksi, eli yhdistävä pelaajaa ajaa osumarekisteröinnin, mutta palvelin sitten tarkistaa, oliko osuma laillinen.

Kuva 8. Battlefield 1 -viivekompensaation esimerkki 2. Kuva on otettu ”Battlenonsense”-Youtube- Kanavan tehdyistä kokeista. (5.)

Kuvassa 8 kakkospelaajan viive ylittää 150 ms. Tässä tilanteessa pelaajalla on 157 ms:n viive ja pelin oikeaan yläreunaan tulee ”tähtäämisennakointi”-indikaattori, mikä tarkoittaa, että pelaajan viive on liian korkea ja osumarekisteröinti muuttuu täysin palvelimen puoleiseksi ja kakkospelaajan täytyy ennakoida tähtäämistään osuakseen pelaajaan. Tällä menetelmällä pyritään välttämään osumien myöhäistä rekisteröintiä, kun pelaaja juoksee suojan taakse.

Peliä suunniteltaessa kannattaa pitää mielessä, onko omalle pelille hyödyllistä implementoida viiveen kompensointia. Peleissä, jossa vaaditaan tarkkaa osumarekisteröintiä, kuten fps-moninpelissä, on melkein pakollista toteuttaa viiveen kompensaatiota, jos halutaan, että oma peli pärjää nykymarkkinoilla. (5.)

3 Unreal 4 -pelimoottorin perusteet

3.1 Johdanto Unreal 4 -pelimoottoriin

Unreal on pelimoottori, jota voi käyttää monenlaisen pelin kehittämiseen. *Unreal* tarjoaa täydellisen version pelimoottorista käytettäväksi täysin ilmaiseksi, kunnes peli alkaa tuottamaan rahaa. Tämän jälkeen Unreal 4 -pelimoottorin kehittäjät (”Epic Games”) ottaa kaikista tuotoista 5 %. Pelimoottori on saanut viime vuosina enemmän suosiota, varsinkin AAA- ja indiepelikehittäjien piireissä. Pelit kuten ”Playerunkown’s Battlegrounds”, ”Street Fighter V” ja ”Gears of War 4” ovat vain osa monista peleistä, joita on kehitetty kyseisellä moottorilla. *Unreal 4* saatetaan valita *Unityn* sijaan, muun muassa C++ -kielen takia, koska pelimoottorissa on enemmän niin sanottuja valmiiksi tehtyjä toteutuksia, jotka mahdollistavat kehittäjiä keskittymään pelkästään pelin kehittämiseen. Moottori on myös koko kehitystiimiin workflow’lle hyödyllinen, muun muassa, visuaaliskriptauskieli mahdollistaa myös artisteja ja pelinsuunnittelijoita toteuttamaan asioita itse pelimoottorin sisällä vähäisellä ohjelmointitaidolla.

3.2 Blueprintit

Blueprintit ovat *Unreal 4* -tekijöiden kehittämä visuaalinen skriptauskieli, jolla voidaan toteuttaa pelitoiminnallisuuksia pelimoottorin sisällä. Konsepti perustuu solmupohjaiseen (”node”) käyttöliittymään, jossa yhdistelemällä valmiiksi rakennettuja solmuja voidaan toteuttaa nopeasti toimivia peliprototyyppejä. Blueprintillä voidaan luoda jopa kokonaisia pelejä, jos pelin tehokkuusvaatimukset eivät ole vaativia. (11.)

Blueprintin-skriptaus toimii teknisesti niin sanotulla ”virtuaalikoneella”, mitä tarkoittaa, että skriptaus toteutukset eivät käänny assembly-kieleksi niin kuin tyypilliset ohjelmointikielet. Skriptattu toteutus käännetään väliesitysmuotoon, jota pystytään ajamaan millä tahansa yhteensopivalla laitteella. Esimerkiksi sama blueprint-koodi on yhteensopiva PC:lle, Mac:lle, Linux:lle, Playstation4:lle ja niin edelleen. Tämän jälkeen koodi käännetään kyseisen laitteen natiivikielen muotoon.

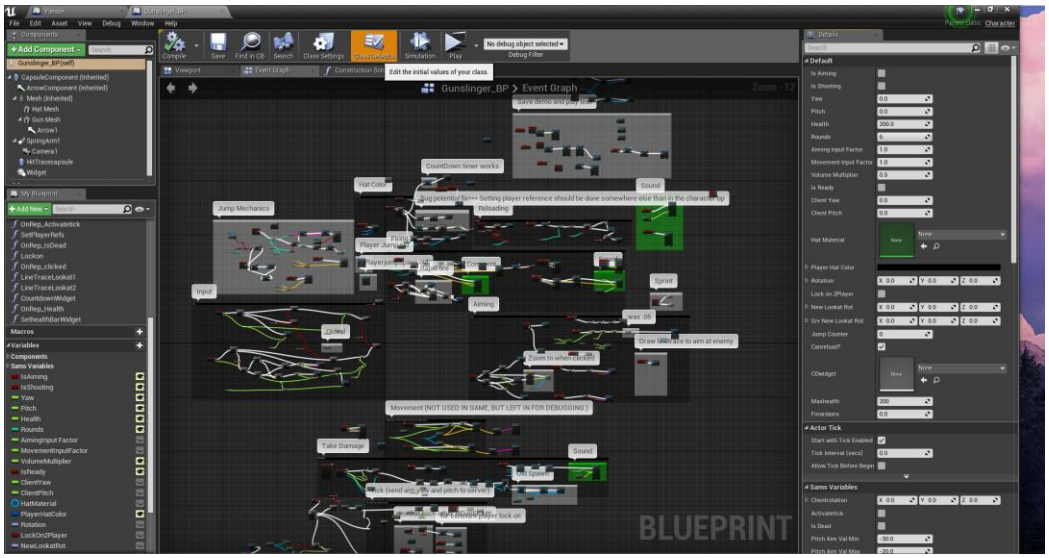
Blueprint-skriptaus mahdollistaa uusien toteutuksien nopeata testausta. Se vähentää monta vaihetta, jotka olisivat tyypillisesti läsnä tavallisen koodikielen toteutuksissa. Haittapuolia on kuitenkin se, että vastaavat toteutukset ovat huomattavasti hitaampia kuin

C++ -kielellä toteutetut, noin 10-15 kertaa hitaammat. Hitaus kuitenkin ilmenee vain, kun toteutuksien kutsumista toistetaan useita kertoja. Skriptausta siis kuuluisi käyttää lähinnä yksittäisten tapahtumien käsittelyssä eikä suuren datamäärän käsittelyssä. Blueprint-skriptausta on kuitenkin mahdollista kääntää C++ -koodiksi (ei ihmisten luettavaksi) - tätä toimintoa kutsutaan ”blueprint nativizationksi”. Tämä toiminto on kuitenkin hätäratkaisu monessa tapauksessa, ja jos peli on toteutettu skriptauskielellä, tämän ei pitäisi olla tarpeellista, sillä peli on todennäköisemmin melko yksinkertainen ja tietokoneelle vaatimaton suoritustehon kannalta. (19.)

Näistä asioista huolimatta blueprint-skriptaus on mainio tapa päästä nopeasti toteuttamaan pelejä ja oppimaan moottorin käyttöä. Skriptaus-kielen oppimisesta on paljon hyötyä myös Unreal C++ -kielen oppimisen suhteen, sillä niiden logiikat ovat suhteellisen yhtenäisiä.

3.2.1 Blueprint-tyypit ja käyttöliittymä

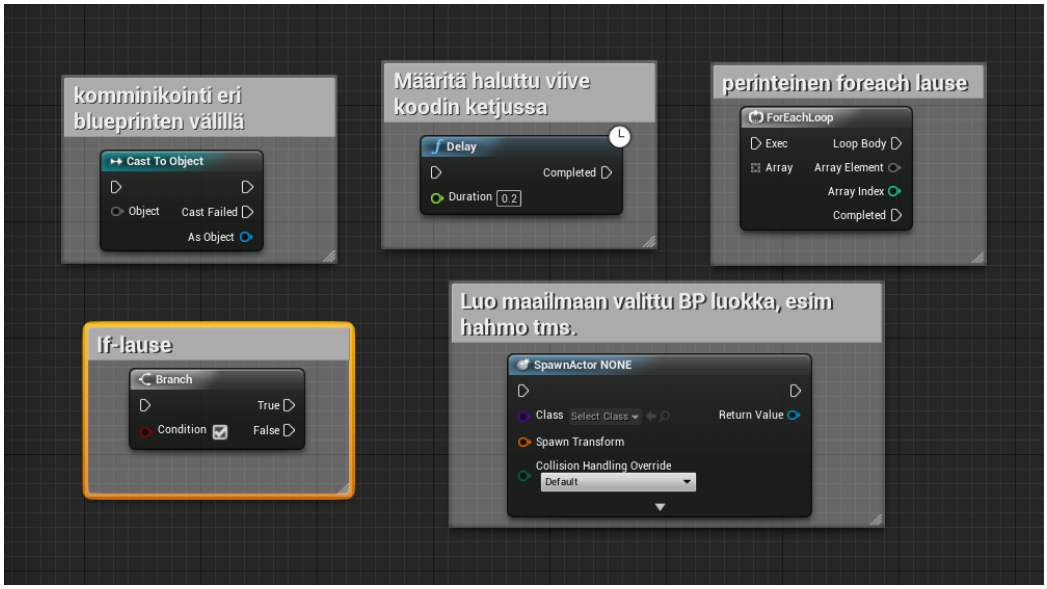
Blueprint-tyyppejä on erilaisia, joista aloittelijoille tärkeimmät ovat muun muassa taso-blueprint (”Level blueprint”) ja blueprint-luokka (”blueprint class”). Blueprint-luokkia on monenlaisia. Käytetyimpiä luokkia ovat hahmo-blueprint (”character blueprint”) ja näyttelijä-blueprint (”Actor blueprint”). Kaikille blueprint-tyypeille on ominaista samankaltainen käyttöliittymä, jonka sisällä voidaan luoda ja määrittää muuttujia (”Variables”), tapahtumia (”Events”), funktioita (”Functions”) ja makroja (”Macro”). Käyttöliittymän sisällä on myös esillä blueprint-tyypin oletusasetukset (”Default class settings”), joilla voidaan esimerkiksi asettaa muuttujien oletusarvoja ja vaihtaa tekstuureita. Käyttöliittymä sisältää tapahtumagraafin (”Event graph”), jossa toteutetaan pääsääntöisesti blueprintin logiikkaa. Blueprint-luokilla on oma konstruktoriskripti (”Construction script”), jossa voidaan vaihtoehtoisesti alustaa muuttujia ennen blueprintin luontia pelissä. Esikatseluikkunasta voidaan havainnoida, miltä esimerkiksi blueprintin hahmo tai actor tulee näyttämään pelissä. (11.)



Kuva 9. Blueprint-käyttöliittymä hahmo-blueprintissä.

3.2.2 Solmut, funktiot ja tapahtumat

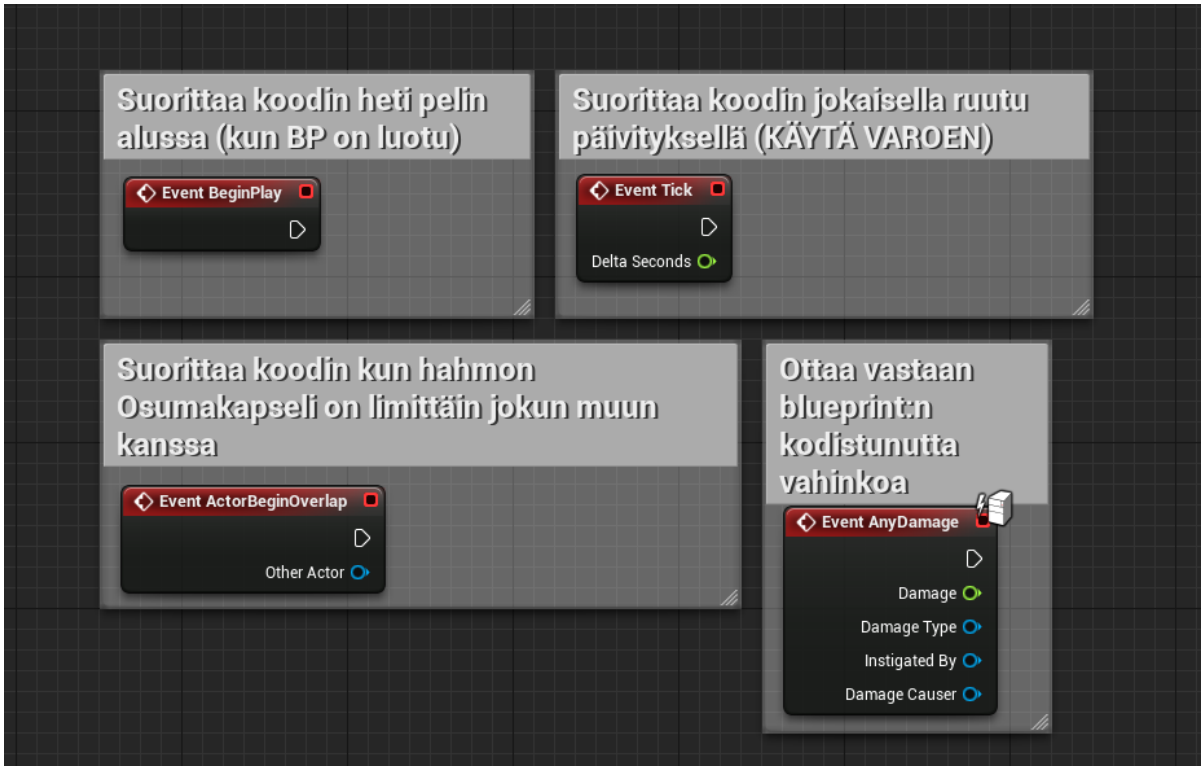
Blueprinttiin saadaan implementoitua toiminnallisuuksia yhdistelemällä yhteen muuttujia (”Variables”), solmuja (”Nodes”) ja tapahtumia (”Events”). Unreal 4 -pelimoottori tarjoaa useita valmiita solmuja, joilla saa kattavasti erilaista logiikkaa toteutettua. Seuraavassa kuvassa käy ilmi olennaisia solmuja, joita tulee usein käytettyä blueprint workflow’n aikana. (18.)



Kuva 10. Esimerkki solmuista blueprintissä.

Usein kun tehdään isompia projekteja, blueprintin sisältö voi kasvaa lukemattomaksi ”spagetiksi”. Luomalla funktioita voidaan vähentää ns. spagetin määrää, varsinkin jos kyseinen toteutus vaatii monta solmua ja samaa toiminnallisuutta halutaan käyttää muualla. On paljon helpompaa kaataa se funktioksi (”Collapse to function”), jota voi katsella omasta ikkunasta kuin yrittää lukea spagetista, mitä tapahtuu koodissa. (15.)

Tapahtumat ovat erittäin olennainen osa blueprint-workflow’ta. Ne määrittävät, milloin koodia kuuluisi ajaa pelin aikana. Unrealin sisällä on useita valmiiksi tehtyjä tapahtumia, joilla saadaan aktivoitua logiikkaa. Hyödyllisiä tapahtumia hahmon blueprintin sisällä ovat muun muassa ”Event begin play”, ”Event any damage”, ”Event tick” ja ”Event overlap”. (16.)



Kuva 11. Esimerkki tapahtumista.

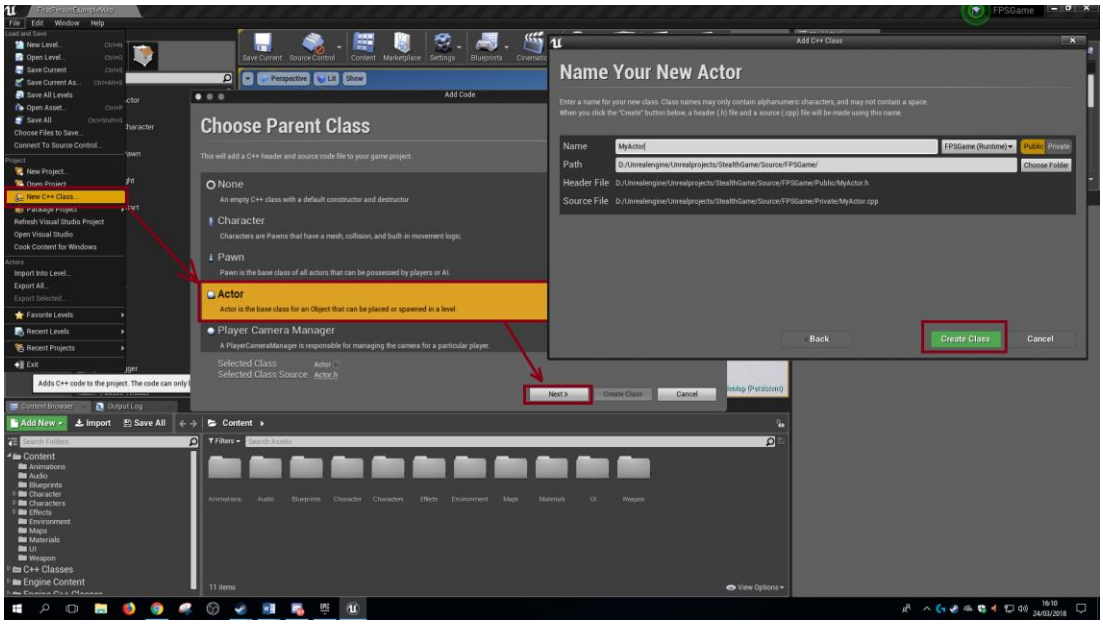
Blueprintissä on mahdollista luoda mukautettuja tapahtumia (”Custom event”), joita voidaan kutsua missä tahansa koodin ketjun aikana. Käsittelen niiden käyttöä tarkemmin moninpeliaiheen osuudessa, sillä ne ovat erittäin olennainen osa blueprint-moninpelitoiminallisuuksien toteuttamisessa.

3.3 Unreal C++ -perusteet

Unreal 4 -pelimoottorista saa täyden tehokkuuden irti, kun ottaa käyttöön C++ -kielen pelien kehittämisessä. Unreal C++ -implementaatio eroaa tyypillisestä C++ -implementoinnista, sillä moottorin sisällä on paljon valmiiksi tehtyjä makroja, funktiota, muuttujia ja luokkia, joita voi hyödyntää. Unreal 4 -pelimoottorin kehittäjät kuvailevat Unreal C++ -kieltä ”avustetuksi” C++ -kieleksi. Kehittäjät voivat siis keskittyä itse pelin tekemiseen, eikä heidän tarvitse huolehtia matalan tasoisista implementoinneista, jos he niin valitsevat. Pelimoottoria voi tietysti muokata oman tarpeen mukaan, jos haluaa, mutta se on harvinaista ja todennäköisesti tarpeetonta ellei ole isossa pelifirmassa, jossa ohjelmoijien määrä ja taito riittävät. Kehittäjät ovat suunnitelleet moottoria niin, että tehokkain tapa käyttää moottoria eli ”workflow” olisi ensin implementoida C++ -luokkia, joita voidaan myöhemmin moottorin sisällä laajentaa blueprinteiksi. Tässä tapauksessa en tarkoita itse skriptaus-kieltä, vaan itse blueprint-objektin luomista. Tarkoitus olisi myös, että uusia pelillisiä toiminnallisuuksia voitaisiin ensin toteuttaa skriptaamalla ja sitten tarpeessa implementoida Unreal C++-kielellä. Ohjelmoijat voivat käyttää haluamansa IDE-ohjelmaa, mutta tyypillisesti päätyvät valitsemaan Microsoftin ”Visualstudion” tai Applen ”Xcode”-ohjelmia.

3.3.1 Luokkien luominen

Omia C++ -luokkia voi generoida Unreal editorin sisällä olevan ”class wizardin” avulla. Esimerkiksi uuden peliolion luominen näyttäisi seuraavanlaiselta.



Kuva 12. Kuva C++ -luokan generoimisesta.

Kuvan 12 näkyy miten C++ -luokka ”MyActor” luodaan. Kun painetaan ”Create Class” -nappia editor avaa visualstudion tai xcoden ja generoi kyseiselle luokalle header- ja cpp -tiedoston.

Luokan header-tiedostoon on valmiiksi luotu sisältöä seuraavanlaisesti.

```
#include "GameFramework/Actor.h"
#include "MyActor.generated.h"

UCLASS()
class AMyActor : public AActor
{
    GENERATED_BODY()

public:
    // Sets default values for this actor's properties
    AMyActor();

    // Called every frame
    virtual void Tick( float DeltaSeconds ) override;

protected:
    // Called when the game starts or when spawned
    virtual void BeginPlay() override;
};
```

Esimerkkikoodi 2. Uuden luodun luokan header-tiedoston valmiiksi generoitu koodi.

Header-tiedostossa luodaan valmiiksi kaksi tapahtumaa, jotka voidaan implementoida cpp-tiedostossa. ”AMyActor” on luokan konstruktori, jossa voidaan alustaa luokan muuttujia. ”BeginPlay:n” avulla voidaan suorittaa koodia, kun luokka luodaan peliin. ”tick” on tapahtuma, jolla voidaan suorittaa koodia jokaisella ruutupäivityksellä.

3.3.2 Ominaisuuksien luonti

Ominaisuuksien kuten muuttujien luonti tapahtuu ”UPROPERTY”-makrolla. Vastaavilla makroilla voidaan myös määrittää, miten esimerkiksi ominaisuutta pystyy käsittelemään editorin sisällä.

```
UCLASS()
class AMyActor : public AActor
{
    GENERATED_BODY()
public:

    UPROPERTY(EditAnywhere)
    int32 TotalDamage;

    ...
};
```

Esimerkkikoodi 3. Esimerkki ominaisuuksien luomisesta C++ -luokan header-tiedoston sisällä.

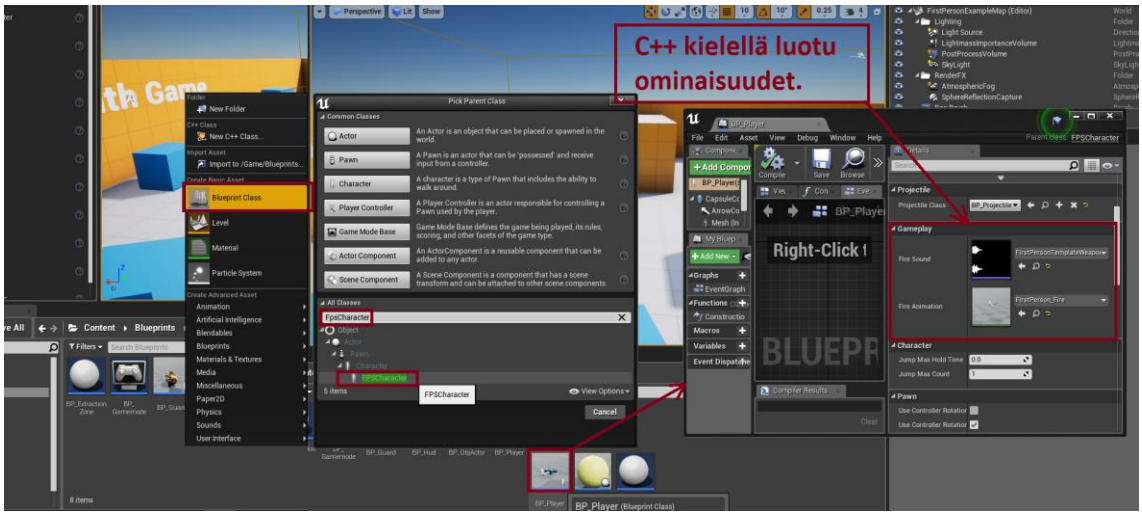
Ominaisuuksille voidaan myös määrittää kategoria ja asettaa niitä blueprint-skriptille editoitavaksi lisäämällä ”UPROPERTY”-parametriin argumentit seuraavalla tavalla.

```
UPROPERTY(EditAnywhere, BlueprintReadWrite, Category="Damage")
int32 TotalDamage;
```

Esimerkkikoodi 4. Muuttujalle tehdään blueprint luettavaksi, Kun välitetään seuraavat argumentit UPROPERTY-makron parametreihin.

Unrealissa on useita eri argumentteja, joilla voidaan säädellä ominaisuuksia. Lisätieto niistä löytyy tämän aiheen merkityssä lähteessä, joka on osa Unreal 4 virallista dokumentaatiota.

Luotuamme halutun ominaisuuden, voidaan sitten kääntää koodia ja avata editori. Editorin sisällä voimme testata uuden ominaisuuden toimivuuden luomalla kyseisestä ”AMyActor”-luokasta blueprint-luokan. Blueprint luodaan kuvan 13 esittämällä tavalla.



Kuva 13. Kuva blueprint-luokan luomisesta.

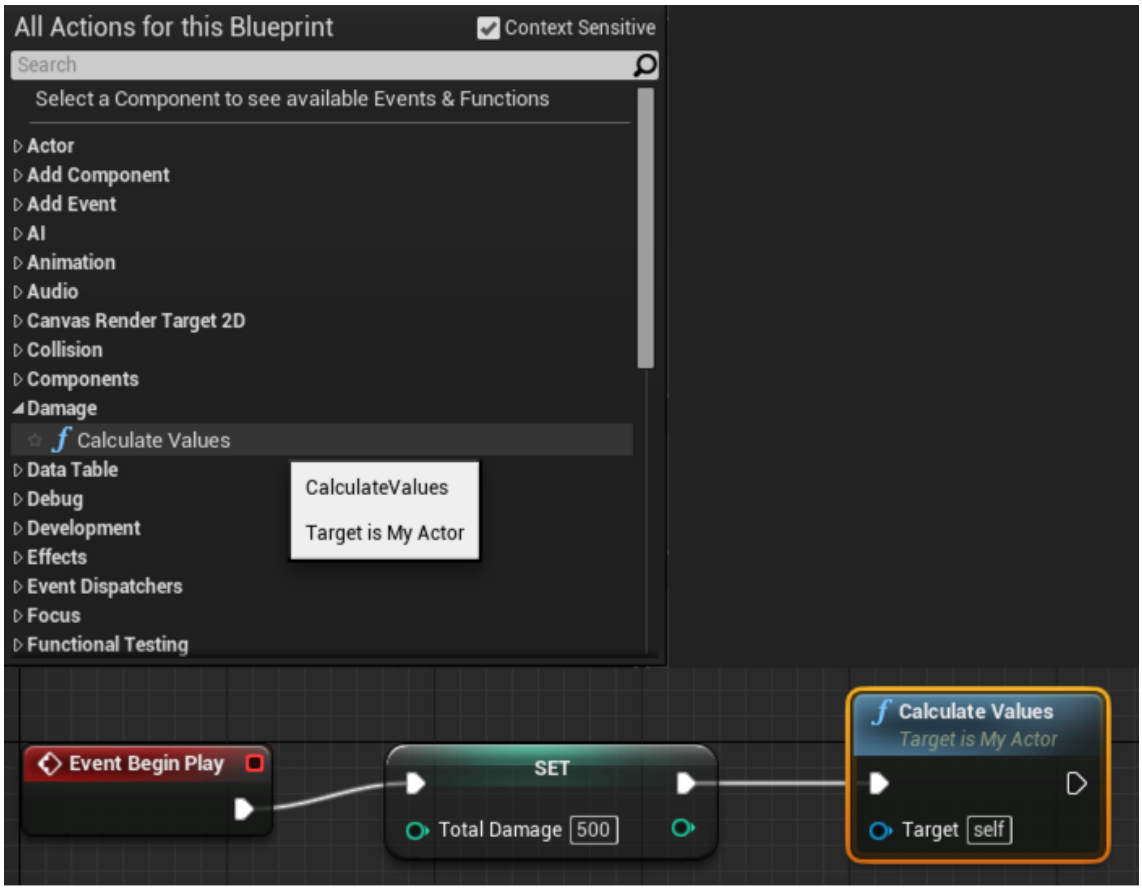
Kuva 13 esittää omasta projektissa luodun ”FPSCharacter”-luokan luomista blueprint -luokaksi. Editorissa oikean hiiren näppäimen avulla avataan valikko, josta valitaan ”blueprint Class”. Tämän jälkeen navigoidaan C++ -kiellä luodun luokan nimen kohdalle ja valitaan kyseinen luokka. Blueprint-luokalle annetaan sille sopiva nimi. Tässä tapauksessa olen antanut nimeksi ”BP_Character”. Kuvan oikealla puolella näkee sitten avatun blueprint-luokan, jossa oikealla ominaisuuksien osiossa näemme uudet luodut ominaisuudet.

3.3.3 Funktion luominen

Unrealissa toimitaan samankaltaisella tavalla, kun luodaan funktioita. Makro, jota käytetään, on nimeltä ”UFUNCTION”. Funktioille voimme myös määrittää voinko sen kutsua editorissa, blueprintissä ja voidaanko sille antaa kategoria.

```
UFUNCTION(BlueprintCallable, Category="Damage")
void CalculateValues();
```

Esimerkkikoodi 5. Kyseisessä koodirivissä luodaan uusi ”CalculateValues”-funktio, jota voidaan kutsua blueprint-luokan sisällä.



Kuva 14. Kuva esittää uuden luodun funktion kutsumista blueprintissä

Kuvassa 14 esitetään, miten uusi luotu funktio voidaan kutsua blueprintin sisällä. ”BlueprintCallable”-argumentilla paljastetaan kyseinen funktio Unrealin blueprint-virtuaalikoneelle, josta blueprint voi kutsua sen.

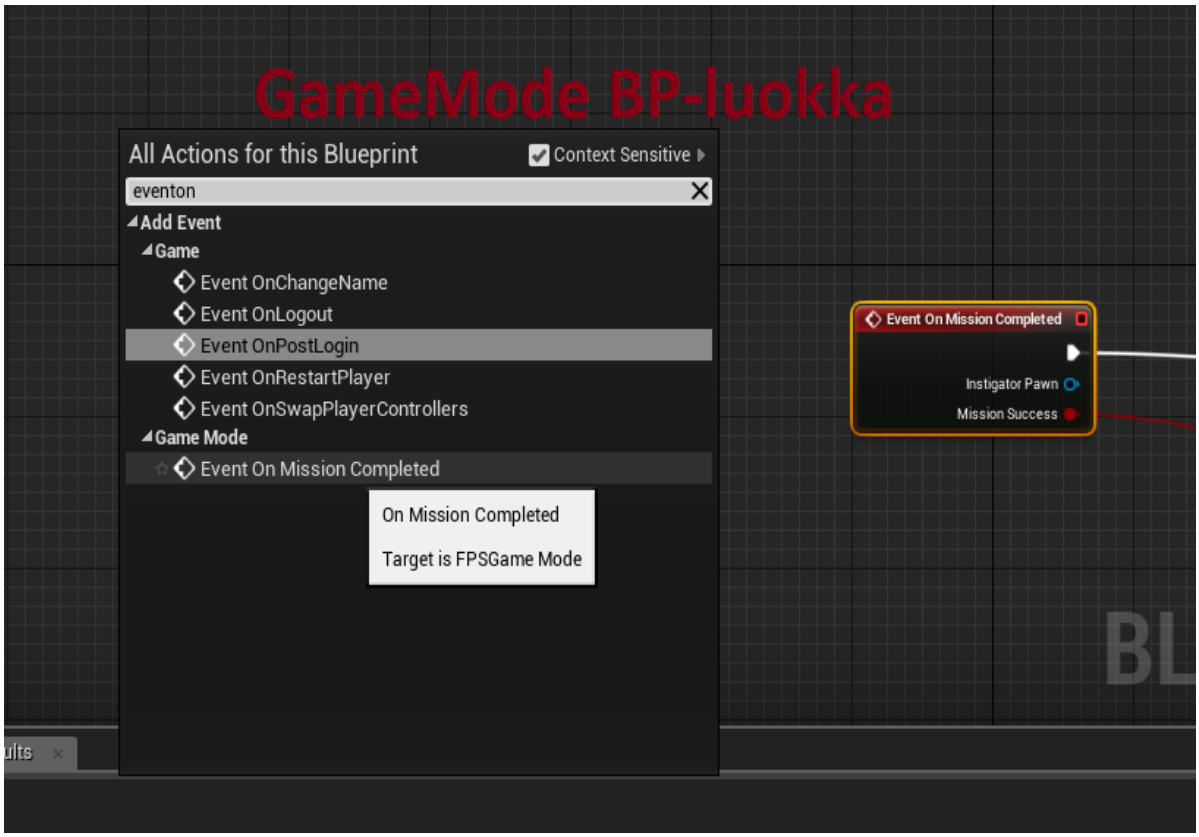
Tulee myös tilanteita, joissa haluamme mahdollisesti toteuttaa funktion blueprint-ympäristössä.

```
UFUNCTION(BlueprintImplementableEvent, Category = "GameMode")
void OnMissionCompleted(APawn* InstigatorPawn, bool bMissionSuccess);
```

Esimerkkikoodi 6. Luokan header-tiedostossa määritetään funktio blueprint-skriptaus-kieleksi implementoitavaksi.

Tämä on mahdollista, kun annetaan UFUNCTION-makron parametriksi argumentti ”BlueprintImplementableEvent”, jolla käytännössä tarkoitetaan, että luodaan uusi

blueprint-tapahtuma kyseiselle luokalle, jota voidaan implementoida blueprint-luokan sisällä esimerkiksi seuraavalla tavalla.



Kuva 15. Esimerkki miten luotu funktiotapahtuma kutsutaan blueprint-skriptin sisällä.

4 Unreal 4 -verkkoarkkitehtuuri

Tässä osiossa käsitellään Unreal 4 -pelimoottorin rakennetta lyhyesti ja sen jälkeen siirrytään verkkorakenteen aiheeseen.

4.1 Moottorin rakenne ja luokat

Ennen kuin päästään itse moninpelitoteutusaiheeseen, on hyvä saada yleisymmärrys Unreal 4 -pelimoottorin rakenteesta ja verkkoarkkitehtuurista. Näiden tietojen avulla on helpompaa välttää logiikkavirheitä toteuttaessa moninpelattavaa peliä.

Unreal 4 -pelimoottorin rakenne koostuu monesta eri luokasta, joita ovat

- peli-instanssi (”Game Instance”)
- pelimuoto (”Game Mode”)
- pelitila (”Game State”)
- peliolio (”Pawn”)
- pelaajan tila (”Player State”)
- pelaajan ohjain (”Player Controller”)
- HUD-luokka (” Heads up display”)
- UMG-luokka (“Unreal motion graphics”).

Unreal käyttää luodessaan uutta projektia kaikkia näitä luokkia. Oletusversioita voidaan vaihtaa tarvittaessa mukautetuksi luokaksi. Jokaiselle luokalle on omia funktioita, solmuja ja tapahtumia, jotka ovat niille ominaisia ja uniikkeja. (13.)

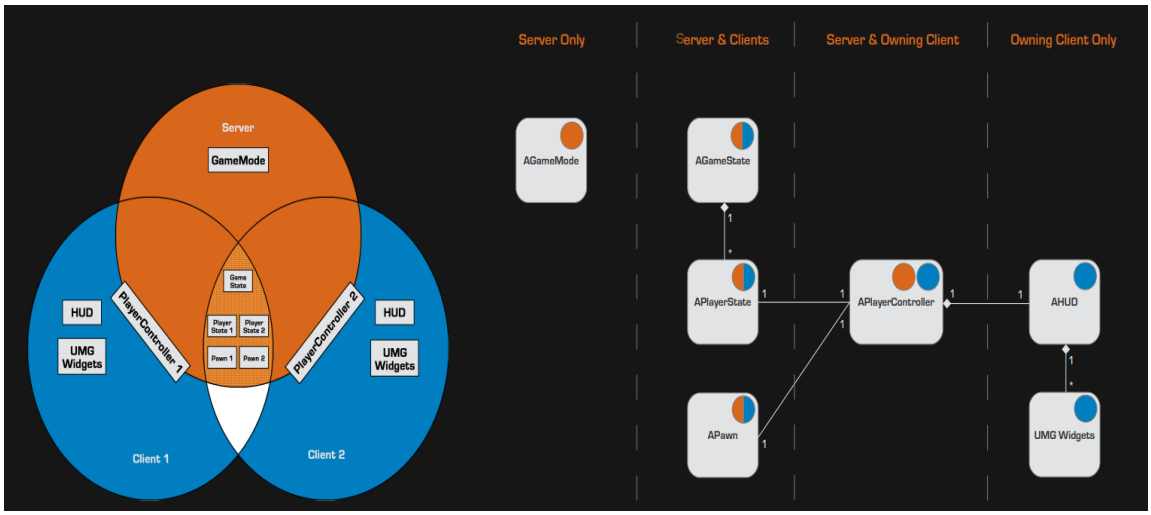
4.2 Tarkemmin luokista

Pelimuoto on yksi Unreal 4 -pelimoottorin keskeisimpiä luokkia. Siellä voidaan määrittää kaikki pelin säännöt. Esimerkiksi jos pelimuotona olisi perinteinen tappopeli, kaikki siihen liittyvä siihen muotoon liittyvä logiikka olisi sen sisällä. Pelintila tekee yhteistyötä pelimuodon kanssa ja ylläpitää tietoa muuttuvista asioista pelin aikana. Pelitilalla on esimerkiksi tiedot pelin alkamisesta ja kauanko peli on ollut käynnissä. Peliolio on pohjaluokka jokaiselle actorille, jota pelaaja tai tekoäly voi ohjata. Sen sisällä voidaan määrittää, minkä näköinen hahmo on ja miten se toimii ja reagoi muuhun pelimaailmaan. Pelaajanohjain on luokka, joka pystyy ottamaan haltuunsa peliolion. Luokka on tärkeä pelimuodoissa, joissa pelaajan hahmoa tuhotaan (useimmissa pelimuodoissa) useasti. Kun hahmo tuhotaan, peliohjain jatkaa olemassaoloaan ja sille määrätään uusi hahmo, jonka se ottaa haltuunsa. (17.)

4.3 Unreal 4 -verkkoarkkitehtuuri

Unreal 4 käyttää ns. ”Client-server” -arkkitehtuuria, joka tarkoittaa, että palvelimella on ”valta” yli kaikkien yhdistävien pelaajien. Tarkemmin kuvailtuna, jos pelaaja haluaa tehdä jotain, se lähettää dataa tehdystä asiasta palvelimelle. Kuvitellaan tilanne, jossa pelaaja haluaa liikkua paikasta toiseen. Tieto lähetetään palvelimelle, joka tarkistaa, onko liike pätevä. Jos se on, se päivittää kyseisen hahmon sijainnin ja välittää siitä tiedon muille pelaajille. On tärkeätä huomioida, että pelaajiin ei voi ikinä luottaa, jotta pelissä

vältyttäisiin huijauksilta. Tästä syystä pelaajat eivät myöskään ikinä kommunikoi keskenään suoraan, vaan tieto välittyy ensin palvelimen kautta. Kuvassa 5 voidaan hahmottaa, miten aiemmin mainitut luokat on sijoitettu verkkorakenteeseen. Tästä käy ilmi se, mitkä luokat ovat palvelimella ja yhdistäville pelaajille olemassa olevia. (10.)



Kuva 16. Luokkien asettelu Unreal 4 -verkon rakenteessa. (10.)

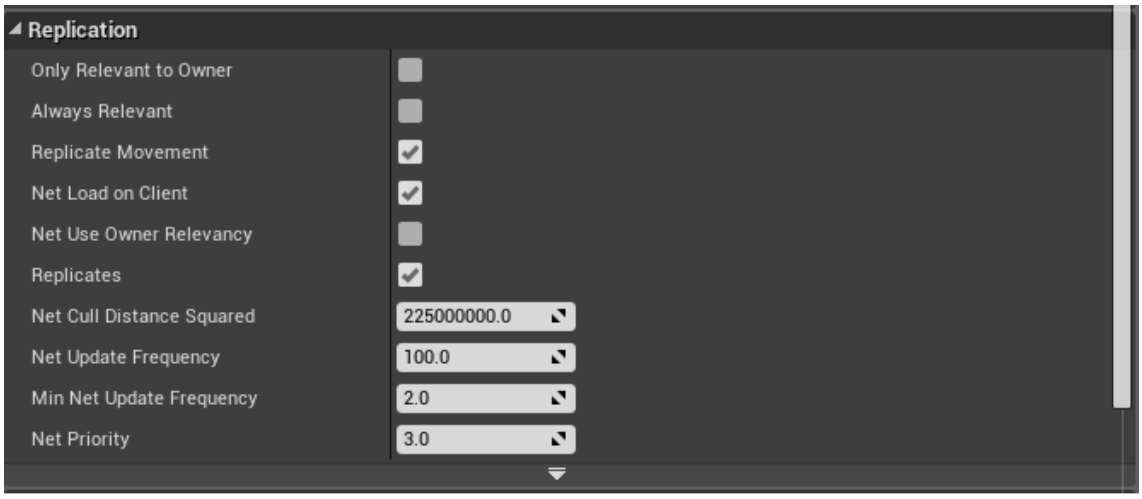
5 Replikaatio- ja moninpelitoteutukset

Tässä työn osiossa käsitellään replikaation käsitettä Unreal 4 -pelimoottorin sisällä. Peleissä replikaatio-käsitteellä tarkoitetaan tiedon jakamista etäiselle pelipalvelimelle tai pelaajalle, jotta se voi välittää tiedon muille palvelimelle yhdistäville asiakkaille. Sen tarkoitus on takaa, että kaikki asiakkaat voivat olla ajan tasalla siitä, mitä pelin tilanteessa tapahtuu, ja se on moninpelien toimivuudelle minimivaatimus.

5.1 Actor-replikaatio

5.1.1 Actor-replikaatio blueprintillä

Unreal 4 -pelimoottorin sisällä on paljon valmiiksi toteutettuja moninpeliomainaisuuksia, joilla pääsee nopeasti alkuun, kun aloittaa moninpelin teon. Olennainen osa Unreal 4 -moninpeliteknologiaa on ns. Actor-replikaatio (”Actor replication”). Kuvassa 6 voidaan hahmottaa, miten helppoa on saada blueprintit moninpelikelpoiseksi.



Kuva 17. Hahmon blueprint oletusasetuksista löytyvä ”replication”-osio.

Kun ”replicates”-lippu on asetettu päälle, kyseinen blueprint on automaattisesti synkronoitu palvelimen ja yhdistävien pelaajien välillä, kun se asetetaan pelattavaan tasoon. Hahmo-blueprintille on ominaista oma liikkumiskomponentti (”Character movement component”), joka on ohjelmoitu moninpelikelpoiseksi. Hahmon liikkeet ovat myös synkronoitu, kun asetetaan ”replicate movement” -lippu päälle. (9.)

5.1.2 Actor-replikaatio C++ -kielellä

Actor-replikaatiota voidaan myös asettaa Unreal C++ -koodilla suoraan luokan konstruktorissa seuraavalla tavalla.

```
//setting the actor replication setting to true.
//uses unreal 4 built in replication system (actor Replication).
SetReplicates(true);
//uses Unreal 4 built in movement replication.
SetReplicateMovement(true);
```

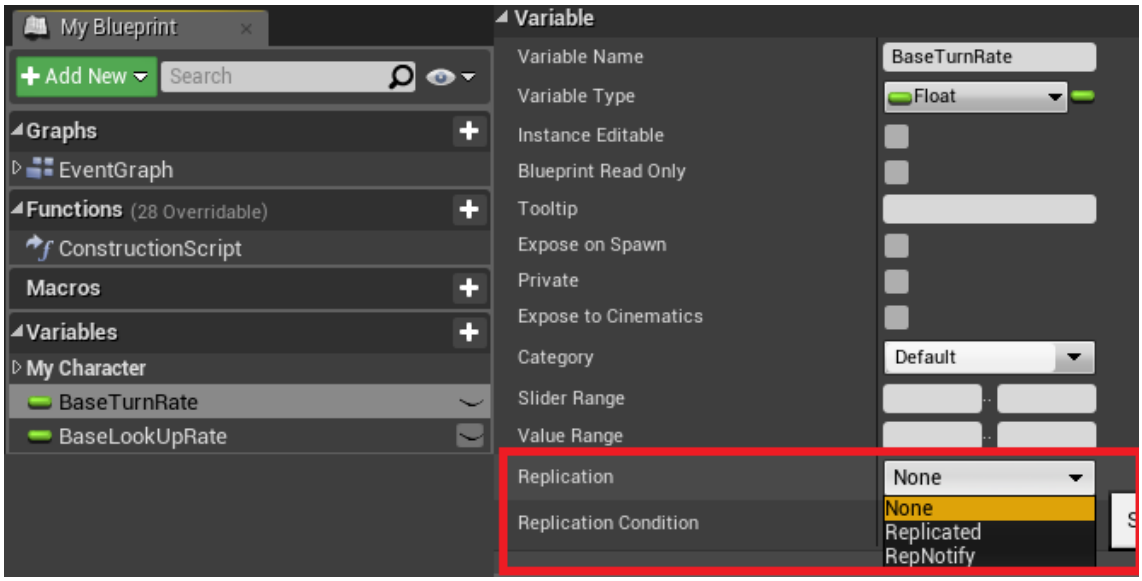
Esimerkkikoodi 7. C++ -luokan konstruktorissa määritetään replikointiasetukset.

Koodi esimerkissä asetetaan suoraan replikaatiofunktiot tukemaan actor-replikaatiota. Saman toteutuksen voi tehdä myös jälkeenpäin, kun luodaan luokasta blueprint -luokka, mutta jos aikoo muuttaa luokkaa vielä myöhemmin olisi kannattavaa laittaa se itse C++ -koodin sisälle tulevien ongelmia välttämiseksi.

5.2 Muuttuja-replikaatio

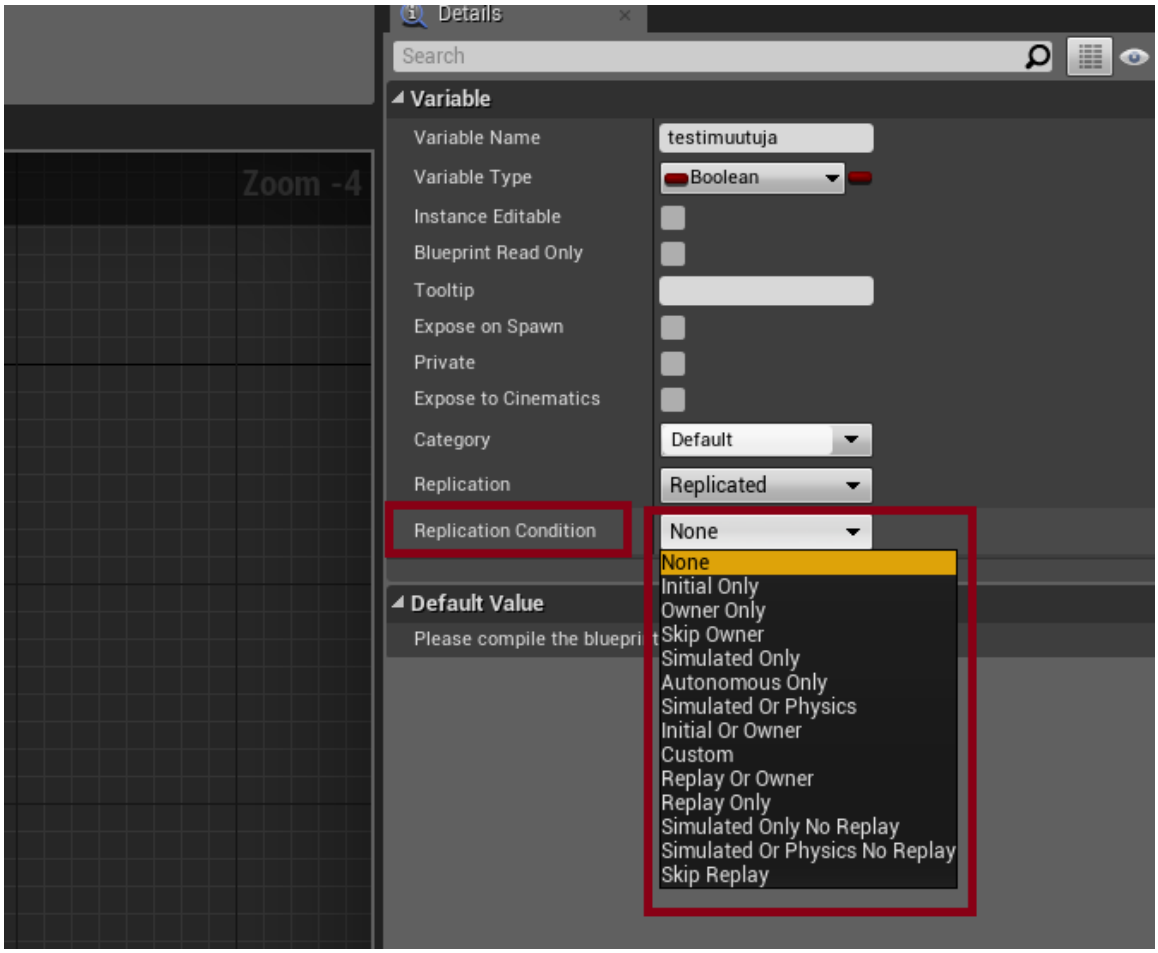
5.2.1 Muuttuja-replikaatio blueprintillä

Blueprintin sisällä olevien muuttujien replikointi onnistuu myös melko helposti. Kun muuttuja luodaan, sen oletusasetuksissa on painike, josta saadaan replikaatio päälle.



Kuva 18. Esimerkki mistä saadaan muuttujan replikaatioasetukset päälle.

Kuten kuvassa 18 näkyy, muuttujilla on kaksi eri replikaatioasetusta. ”Replicated” on yleisin asetus. Sen ollessa päällä muuttujan tiedot päivittyvät automaattisesti, kun sen omistava blueprint itse päivittyy. ”RepNotify” mahdollistaa samat ominaisuudet kuin ”Replicated”-asetus, mutta sen lisäksi se luo erillisen funktion ”On_rep<muuttujan nimi>” kyseiseen blueprintin sisälle. Funktio kutsutaan automaattisesti, kun muuttujan arvo vaihtuu. Tämä on hyödyllinen asetus, kun halutaan päivittää hahmon Hp:ta (tästä esimerkki myöhemmin). ”Replication Condition” -asetuksella voidaan määrittää muuttuja kohtaisia replikaatio-optimointeja. Seuraavassa osiossa käydään läpi tarkemmin eri asetuksia, joita näkyy kuvassa 19. (9.)



Kuva 19. Esimerkki blueprint-luokan replikointin ehtoasetuksista.

5.2.2 Muuttuja-replikaatio C++ -kielellä

Muuttujien tai ominaisuuksien replikointi C++ -kielellä ei ole yhtä suoraviivaista kuin Actor-replikaatiossa, mutta sekään ei ole hankalaa. Esimerkiksi jos luodaan moninpelipelaajalle elopistemuuttuja, se asetettaisiin ensin pelihahmon header-tiedostoon seuraavalla tavalla.

```
//Replicated Health
UPROPERTY(Replicated)
float health;
```

Esimerkkikoodi 8. Pelihahmon header-tiedossa luotu muuttujan replikointi.

UPROPERTY-makrolle välitetään ”Replicated”-argumentti parametrina. Parametreihin voidaan myös lisätä kategoria, tai se voidaan asettaa blueprint-luettavaksi ja

editoitavaksi samalla tavalla kuin miten aikaisemmin on tässä työssä mainittu. Tämän jälkeen siirrytään pelihahmon Cpp-tiedostoon ja kirjoitetaan seuraavat koodirivit.

```
void AFPSCharacter::GetLifetimeReplicatedProps(TArray< FLifetimeProperty > & Out-
LifetimeProps) const
{
            Super::GetLifetimeReplicatedProps(OutLifetimeProps);
            //asetetaan lista muuttujista joita halutaan replikoida ja niille
            ehdot halutessa.
            DOREPLIFETIME(AFPSCharacter, health);
            DOREPLIFETIME(AFPSCharacter, bIsCarryingObjective);
}
```

Esimerkkikoodi 9. Pelihahmon Cpp-tiedostossa kutsuttu “GetLifetimeReplicatedProps”, jolla määritetään kyseisen luokan muuttujien replikaatiota ehtoineen halutessa.

Tietyssä tilanteissa on järkevää asettaa ehtoja muuttujan replikoinnille, sillä jos replikoisimme aina kaikki pelin muuttujia tai ominaisuuksia täydellisesti, se voi hidastaa pelin suoritusta ja pahimmassa tapauksessa aiheuttaa menetettyä dataa tai suurta lagia. Ratkaisuksi tähän löytyy erinäisiä ehtoja, joilla voidaan optimoida koodia.

Replikoinnille ovat olemassa seuraavia ehtoja.

- **COND_InitialOnly**, joka voidaan asettaa ominaisuuksille, jotka replikoidaan vain kerran pelin alussa kaikille pelaajille. Jokainen myöhässä oleva pelaaja saa päivitetyn replikoidun tiedon kerran, kun he liittyvät peliin.
- **COND_OwnerOnly**, päivittää replikoidun ominaisuuden vain omistavalle actorille.
- **COND_SkipOwner**, lähettää replikoidun ominaisuuden kaikille muille paitsi omistavalle actorille.
- **COND_SimulatedOnly**, lähettää replikoinnin tiedon vain actoreille, joita simuloidaan paikallisesti pelaajan koneella palvelimen antaman tiedon perustella.
- **COND_AutonomousOnly**, välittää replikoidun tiedon kaikille actoreille, joilla on omistajuus actoriin ja joita ohjataan. Tyypillisesti actorille jonka pelaajanohjain- luokka on ottanut haluunsa (Posessed) eli kyseistä actoria voidaan ohjata pelaajan tai tekoälyn toimesta.
- **COND_SimulatedOrPhysics**, välittää replikoinnin actoreille, jotka ovat simuloitu tai joilla on extra replikoidun fysiikka asetus muutujja päällä. (kulma nopeudesta kertovaa dataa.)
- **COND_InitalOrOwner**, välittää replikoidun ominaisuuden kerran pelin alussa tai jatkuvasti omistavalle actorille.
- **COND_Custom,** joilla voidaan määrittää itsetehty ehto.

Replikointi ehto lisätään DOREPLIFETIME_CONDITION-makron viimeiseen parametriin esimerkiksi seuraavalla tavalla. (9; 21.)

```
DOREPLIFETIME_CONDITION(AFPSCharacter, bIsCarryingObjective, COND_OwnerOnly);
```

Esimerkkikoodi 10. Muuttuja ominaisuuden optimointi ”COND_OwnerOnly” argumentin välittämisellä makron parametriin.

Unreal C++ -kielessä voidaan myös toteuttaa aikaisemmin mainitut ”Rep_notify”-muuttujia, jotka päivittävät replikoidun muuttujan tiedot, kun itse muuttujan tai ominaisuuden tieto vaihtuu. Tästä näytän esimerkin myöhemmin moninpelitoteutuksien osion yhteydessä.

5.3 Tapahtuma-replikaatio

5.3.1 Tapahtuma-replikaatio blueprintillä

RPC:llä (”Remote procedure call”) tarkoitetaan funktiota, jota kutsutaan paikallisella koneella, mutta sama funktio suoritetaan etäisellä palvelimella. Blueprint-ympäristössä on mahdollista toteuttaa RPC-funktioita mukautetuilla tapahtumilla (”Custom event”) ja asettamalla ne replikoitumaan.

Kuva 20. Mukautettu tapahtuma ja sen ominaisuudet.

Tapahtuman replikaatio saadaan päälle samankaltaisesti kuin muuttujissa ja actorissa, mutta kuten kuvassa 8 näkyy, sen asetukset eroavat niistä. Tapahtumassa on kolme erilaista replikaatiovaihtoehtoa: ”Run on server”, ”Multicast” ja ”Run on owning client”.

Vaihtoehtojen nimet voivat tietyissä tilanteissa olla harhaanjohtavia. Lopputulokseen vaikuttaa, kuka omistaa kyseisen blueprintin ja kuka kutsuu tapahtuman.

”Run on server” on pääsääntöinen tapa välittää tietoa pelaajalta palvelimeen. Jos palvelin kutsuu tapahtuman, se suoritetaan ainoastaan palvelimella. Kun pelaaja omistaa actorin ja kutsuu tapahtuman, se replikoituu myös palvelimelle.

”Multicast” on tapa suorittaa tapahtuman palvelimella ja välittää tiedon siitä kaikille muille pelaajille riippumatta siitä ovatko he kyseisen actorin omistajia. Toisaalta, jos se kutsutaan pelaajan puolelle, tapahtumaa kohdellaan niin kuin se ei olisi replikoitu ollenkaan.

”Run on owning client” suorittaa tapahtuman omistavalle pelaajalle, kun se kutsutaan palvelimella. Kun se kutsutaan pelaajan puolella, sitä kohdellaan samankaltaisesti kuin ”Multicast”, eli tavallisena ei-replikoituneena funktiona. (9.)

5.3.2 Tapahtuma-replikaatio C++ -kielellä

Replikoituja tapahtumia voidaan luoda myös Unreal C++ -funktioina. Funktioiden replikointi seuraa samoja periaatteita kuin blueprint-skriptauksessa ja niiden implementointi on suhteellisen helppoa. Ensimmäiseksi pitää lisätä omaan projektin header-tiedoston yläpuolelle ”UnrealNetwork.h” Include-kirjasto. Itse replikaatioasetus määritellään UFUNCTION-makron parametrin argumenttina esimerkiksi seuraavalla tavalla.

```
//Serverside fire function replication
UFUNCTION(Server, Reliable, WithValidation)
void ServerFire();
```

Esimerkkikoodi 11. Koodiesimerkki ampumismekaniikan, funktion replikointiasetuksen alustuksesta luokan header-tiedostossa.

Itse funktio implementoidaan luokan Cpp-tiedostossa, funktioon kuitenkin kirjoitetaan loppuosaan ”_Implementation” seuraavalla tavalla.

```
void AFPSCharacter::ServerFire_Implementation()
{
      // Yritä ampua luotia aseesta.
}
```

Esimerkkikoodi 12. Cpp-tiedoston funktio implementaatio esimerkki ampumismekaniikasta.

Tämän jälkeen pitää vielä tarkistaa liikkeen pätevyyttä pelipalvelimelta. Palvelin tarkistaa, onko kyseisissä RPC-funktioissa huonoja tai epäpäteviä parametreja. UFUNCTION-makrossa olevalla argumentilla ”WithValidation” tarkoitetaan juuri sitä, että halutaan tarkistaa RPC-kutsun pätevyyttä. Tarkistusfunktio voi määrittää esimerkiksi seuraavanlaisella tavalla.

```
bool AFPSCharacter::ServerFire_Validate()
{
            return true;
}
```

Esimerkkikoodi 13. Esimerkki palvelin tarkistus funktioista luokan Cpp-tiedostossa.

Kun tarkistus funktio palauttaa ”true:n”, RPC-funktio koetaan päteväksi ja se suoritetaan. Kyseiseen funktioon ei ole erikseen määritelty toimintoa, joilla RPC-funktion hylättäisiin. Tämän voisi määrittää esimerkiksi funktiossa, jossa ladataan asetta.

```
bool AFPSCharacter::ServerReload_Validate(int32 addAmmo)
{
            if(addAmmo > MaxAmmoCount)
            {
                        //heittää pelaajan pois pelistä
                        return false;
            }
            //suorittaa funktion
            return true;
}
```

Esimerkkikoodi 14. Semipseudokoodia aseen lataamismekaniikan Rpc-funktion tarkastukseen.

Tässä esimerkissä pelaaja yrittää ladata aseensa. Palvelin tarkistaa, yrittääkö pelaaja ladata lippaaseen enemmän luoteja kuin mitä lipas pystyisi ottamaan. Jos palvelin toteaa, että pelaaja yrittää huijata laittamalla liika luoteja lippaaseen, palvelin katkaisee yhteyden pelaajan ja palvelimen väliltä.

Unreal C++ RPC -funktioille voidaan myös määrittää replikointitapa samalla tavalla, miten blueprint-tapahtumia replikoidaan. UFUNCTION-makroon voidaan laittaa parametriksi ”Client”, ”NetMulticast” tai ”Server”, jotka toimivat samalla tavalla, kun miten aikaisemmin kuvailtiin tapahtumien RPC-funktiossa. Jokaiselle RPC-funktioille voidaan myös määrittää luotettavaa ”Reliable”- tai epäluotettavaa ”Unreliable” -suoritusta. (9.)

Tässä vielä tiivistelmä onnistuneen RPC-funktion- tai tapahtuman luomisen:

- RPC-funktio voi kutsua vain actoreista.
- Actorin täytyy olla replikoitu, joko blueprintin tai C++ -luokan sisällä.
- Jos RPC-funktio kutsutaan palvelimella yhdistävän pelaajan käytettäväksi, RPC suoritetaan ainoastaan kyseisen actorin omistavalle pelaajalle.
- Vastaavasti jos RPC kutsutaan yhdistävällä pelaajalla palvelimelle käytettäväksi, yhdistävän pelaajan täytyy omistaa kyseisen actorin, jolla RPC kutsutaan.
- Multicast RPC poikkeaa niin, että jos se kutsutaan palvelimella, funktio suoritetaan paikallisesti ja myös kaikille yhdistäville pelaajille. (9.)

5.4 Replikaation priorisointi

Moninpelien kehittämisessä liittyy myös monelle pelaajalle piilossa olevia tai huomaamatta jääneitä asioita. Konkreettinen esimerkki tästä on, miten myöhässä tai myöhemmin yhdistävän pelaajat pitävät ajan tasalla tietonsa pelissä tapahtuvista asioista, tai mistä tiedosta pelaajan kuuluisin olla tietoinen, jotta pelin suorituskyky tai nettiyhteyden siirtonopeus ei ylikuormittuisi. Nämä haasteet tulevat kaikissa moninpeleissä, mutta ilmenevät erityisesti MMO-peleissä.

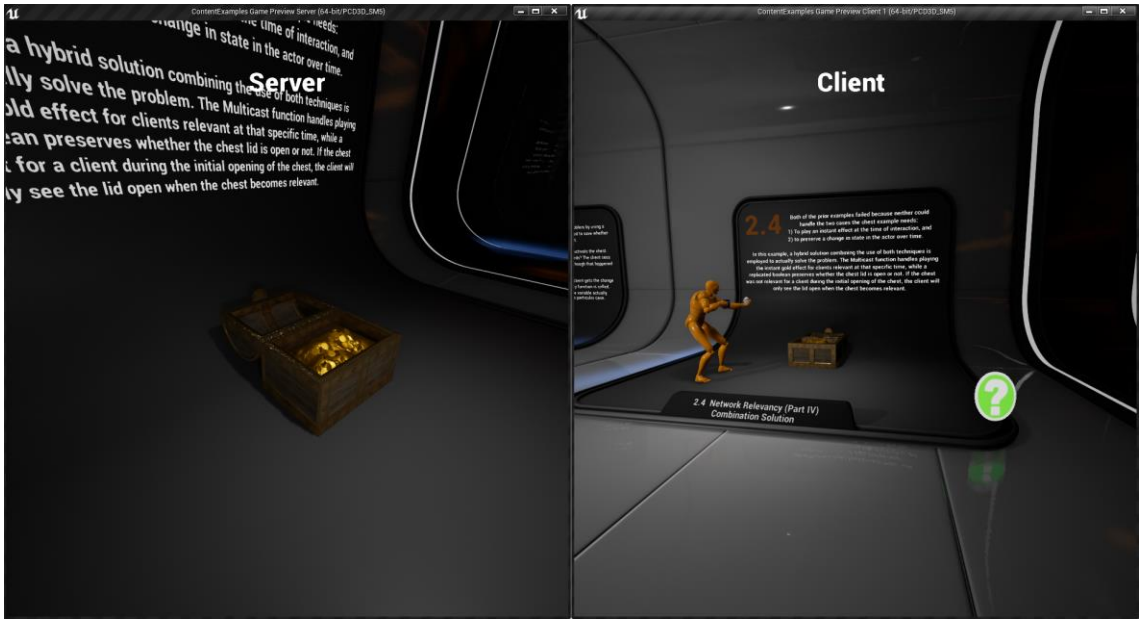
Kuvitellaan MMO-peliä, jossa maailmankartta on monen hehtaarin kokoiseksi luotu, jossa pelaajia voi olla monta tuhatta pelaamassa samaan aikaan. Jos jokainen yhdistävä pelaaja päivittäisi jokaikisen muun pelaajan replikoidut tiedot jatkuvalla syötöllä, pelistä tulisi nopeasti diaesityksen kaltainen kokemus, jos peli edes pyörisi tässä vaiheessa. Ratkaisuksi tähän on niin sanottu replikaation priorisointi tai englanniksi kutsuttu ”Network Relevancy”. Termillä tarkoitetaan sitä, että rajoitetaan ja optimoidaan jokaisen yhdistävän pelaajan välitettyä ja välitettävää tietoa. Toisin sanoen luodaan maksimietäisyys minkä päähän pelaaja päivittää replikoitua tietoa maailmasta. Asian voi kuvitella tietynlaisena ”replikaatio LOD” (”Level of detail”) ominaisuutena. Otetaan tarkasteluun Unreal 4 ”networked features” -projekti esimerkkiä.

Kuva 21. Kuva ”Networked features” -projektista, esimerkki miten actoreiden toiminnallisuuksia voidaan rajoittaa, optimoimaan nettipeliä.

Ensimmäisessä kuvaesimerkissä palvelimen puolella oleva pelaaja (”Server”) avaa kirstun. Sen toiminnallisuuteen on käytetty ”multicast”-replikaatiotapahtumia, joita on määritelty avaavan kirstun ja samanaikaisesti toistavan partikkeliefektin. Yhdistävä asiakas (”Client”) seisoo kyseisen kirstuun määriteltyyn ”olennaisuus”- tai ”relevancy”-alueen ulkopuolella (kirstun ympärillä oleva sininen ympyrä). Yhdistäväpelaaja ei siis näe tässä tapauksessa kirstun avaamista lainkaan, koska hän oli kyseisen alueen ulkopuolella tapahtuman hetkellä. Kuvan esittämässä esimerkissä on myös toinen ongelma, sillä jos yhdistäväpelaaja astuisi alueen sisälle tapahtuman jälkeen, hänelle ei välittyisi tieto, että kirstu olisi avattu ja kirstu näkyisi edelleen suljettuna.

Kuva 22. Kuva ”Networked features” -projektista, esimerkki missä yritetään korjata ongelma, jossa yhdistävä pelaajalle päivitetään olemassa oleva tieto kirstun avaamisesta.

Toisen kuvan esimerkissä näkyy tilanne, missä yhdistäväpelaaja kävelee ”relevancy”-alueen sisälle. Aikaisemmin mainittua ongelmaa on yritetty ratkaista lisäämällä ”Rep_notify”-muuttuja replikaatiototeutusta koodiketjuun. Toisin sanoen, kun kirstu avataan, muuttuja päivittyy, muuttujan päivityksen myötä suoritetaan RPC-tapahtumat pelaajille. Pääasiassa yhdistävä pelaaja saa nyt tiedon oikein. Kirstu on siis avattu aikaisemmin toisen pelaajan toimesta. Kuvassa näkyy vielä kuitenkin ongelma, että partikkeliefekti toistuu, kun yhdistävä pelaaja astuu olennaisuus alueen sisälle. Syy tähän on se, että kyseisessä esimerkissä replikoitu muuttuja on vastuussa kirstun avaamisen ja partikkeliefektin toistamisesta.



Kuva 23. Kuva ”Networked features” -projektista, esimerkki jossa ratkaistaan replikointivirheet yhdistelemällä edellisten kuvien tekniikoita yhteen.

Kuvassa 23 esiintyy ratkaistu tilanne, jossa kirstu nyt avataan pelkästään, jos se on yhdistävälle pelaajalle olennainen replikoinnin suhteen. Toteutuksessa on yhdistetty tapahtuma- ja muuttuja-replikaatiota ”Rep_Notify”-boolean-muuttujalla siten, että boolean-muuttuja on pelkästään vastuussa kirstun avaamisesta ja RPC-funktio toistaa partikkeliefektin. Toisin sanoen partikkeliefekti toistetaan vain, kun kirstu avataan ensimmäistä kertaa, mutta kirstun avaamisen tieto välittyy myös myöhemmin pelaajille, joille koetaan sen olevan olennaista.

Actorien replikaatio-osiossa, kuten kuvassa 23 voidaan nähdä, miten replikaatiota koetaan olennaiseksi. Esimerkiksi ”Networked features” -projektin kuvaesimerkeissä olevan

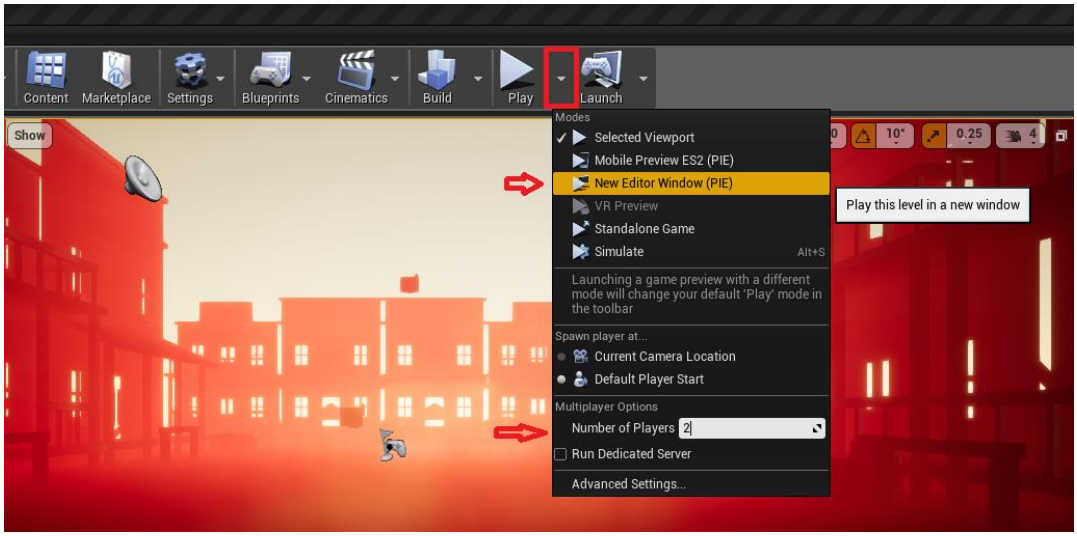
siniset ympyrät on toteutettu hyödyntäen kirstun Actorin ”Net cull Squared” -muuttujaa, jolla määritetään kyseisen actorin replikaation olennaisuutta muille pelimaailmassa oleville actoreille.

6 Esimerkkejä moninpelin toteutuksesta

Tämä osio tulee koostumaan suurimmaksi osaksi kuvista, joissa demonstroidaan moninpelitoteutuksia. Hyödynnän Unreal 4 -kehittäjien ”Multiplayer shootout” -projektia, multiplayer-tutoriaaleja ja oma kehittämäni ”RisingCut”-projektia, kun niissä on paljon hyviä moninpeliesimerkkejä. Kuvassa 24 näytän kuitenkin, miten ylipäänsä pystyy kokeilemaan moninpeliä Unreal 4 -editorin sisällä.

6.1 Perustoteutusmenetelmät ”Multiplayer Shootout” -projektista

6.1.1 Perustoteutus blueprintillä

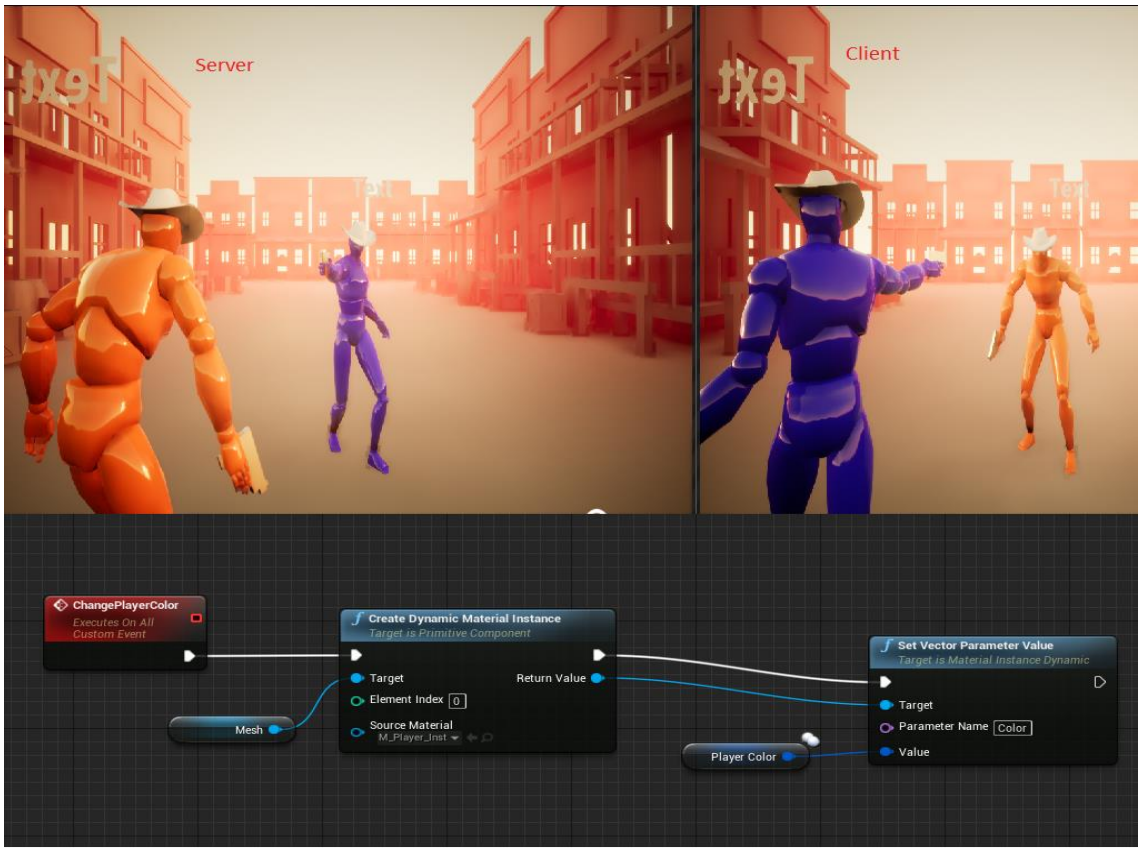


Kuva 24. Esimerkki miten saadaan moninpeli testausasetukset päällä Unreal 4 -editorin sisällä.

Editorin play-napin viereistä nappia painamalla avataan play-asetusvalikko. Pelaajien määräksi laitetaan numero 2 ja vaihdetaan oletusikkunaksi ”New Editor Window” (PIE).

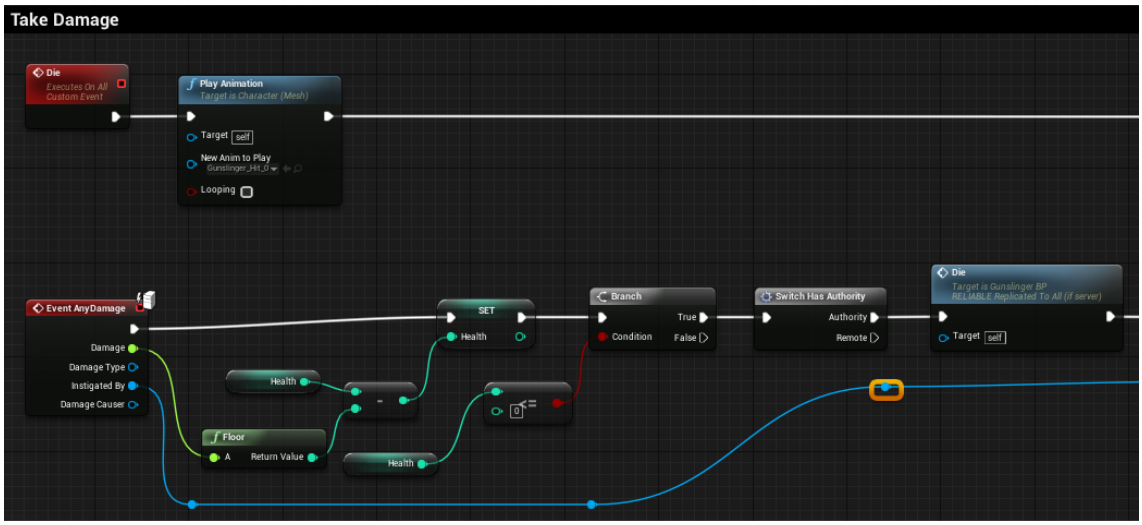
Kuva 25. Esimerkki Haamu-actorin värin muuttamisesta, Hahmon nimi ja väri muuttuu.

Kuvassa 25 on esimerkki hahmon replikaatiosta. Siinä hyödynnetään ”Switch has authority” -makroa, jotta saadaan nimet ja värit muutettua palvelimen ja pelaajan puoleisissa kutsuissa. Toteutuksessa on tarpeen laittaa haamu-blueprintin ”Replicates”-lippu päälle ominaisuusasetuksista, jotta se toimii oikein.



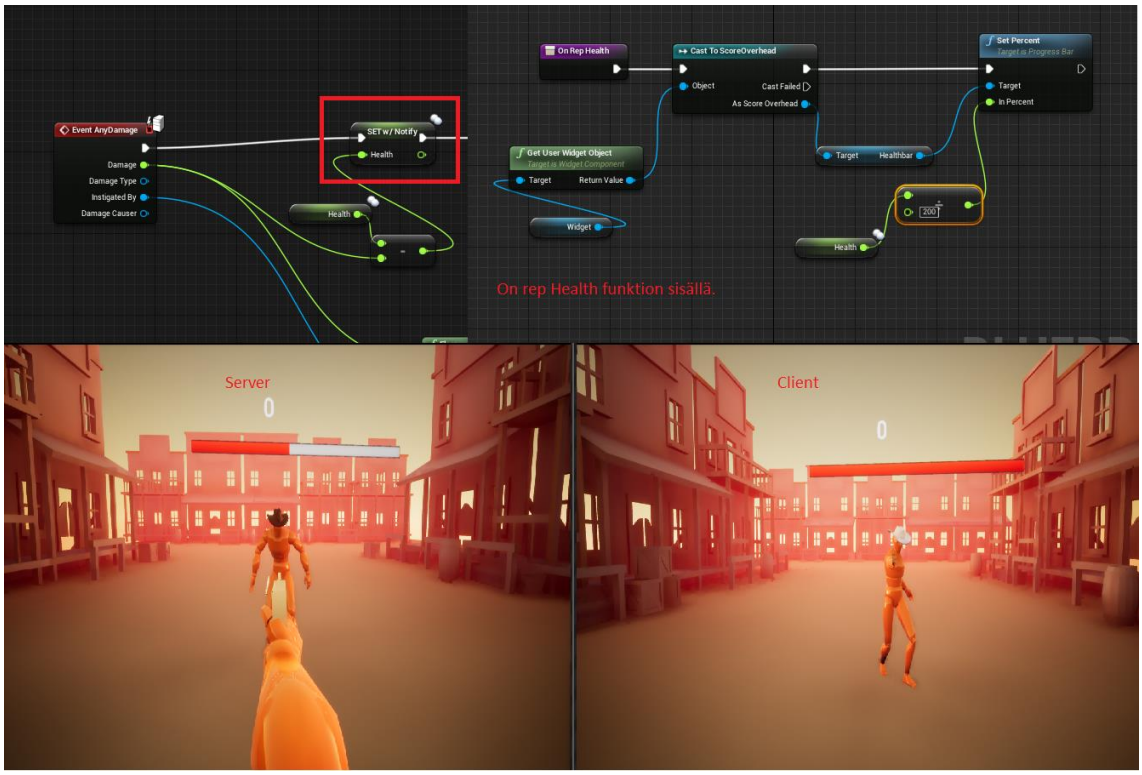
Kuva 26. Esimerkki toisen pelaajan värin muuttamisesta.

Tässä esimerkissä luodaan hahmo-blueprintin sisälle ”Multicast”-tapahtuma, joka kutsutaan pelimuodon sisällä hahmonluonti koodiketjun perään. Aina kun toinen pelaaja luodaan (pelin alussa tai kuoleman jälkeen), se vaihtaa värinsä.



Kuva 27. Esimerkki hahmon vanhingon vastaanottamisesta. Sisältää myös yksinkertaisen kuolema-animaation, kun ”health”-muuttuja on yhtä suuri tai pienempi kuin nolla.

”Event Any Damage” on tapahtuma, joka tapahtuu ainoastaan palvelimella, joten kaikki suoritettu koodi sen jälkeen välittyy palvelimen kautta ensin.”Switch Has Authority” ilmenee vielä varmistuksena, että ”Die”-tapahtuma tapahtuu varmasti palvelimella ensin.



Kuva 28. Esimerkki ”RepNotify”-muuttujan replikaation käytöstä.

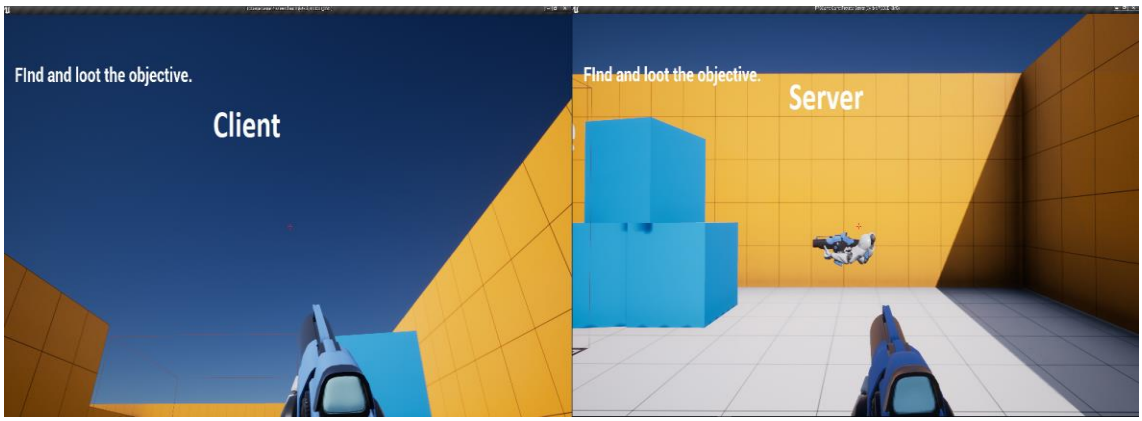
Kuvan 28 esimerkki kuvastaa, miten pelaajan ”health”-muuttuja voidaan asettaa käyttämään ”RepNotify”-asetusta. Kun pelaaja ottaa osuman, ”On Rep Health” -funktio päivittää pelaajan yllä olevaa mittaria.

6.1.2 C++ -kielellä

Seuraavat esimerkit ovat yksinkertaisesta ”Stealth”-peliprojektista. Pelissä olevat ihmispelaajat yrittävät saada haltuunsa objektin ja viedä sen maaliin ilman että kentällä partioivat tekoälyt huomaavat heidät. Tekoäly partioi ympäri kenttää. Kuultuaan pelaajan liikkumisen tai ammutun luodin, ne reagoivat siihen ja kääntyvät siihen suuntaan. Pelaaja

voisi siis hämätä tekoälyä ampumalla luodin johonkin suuntaan. Jos pelaaja nähdään, peli on ohi.

Ensimmäinen esimerkki on hyvin olennainen ongelma liittyen fps-moninpeleihin. Esimerkki on siis, miten pelaajien tähtäysrotaatiot päivittyvät muille pelaajille. Tämä toteutus on harvemmin ensimmäinen asia, mikä tulee pelaajille mieleen, kun he miettivät pelien toiminnallisuuksia ja tätä voidaan pitää itsestäänselvyytenä. Unrealissa hahmojen rotaatiot ovat osittain valmiiksi toteutettuja actor-replikaation kautta. Tarkastellaan seuraavaksi kuvan 29 esimerkkiä.



Kuva 29. Kuva esimerkki fps-moninpelin rotaatiosta ilman actorin kaltevuuden replikoinnin lisäämistä.

Kuvassa 29 näemme, miten asiakas tähtää noin 45 asteen kulmassa taivasta kohti. Palvelimen puolella oleva ohjattava pelaaja onnistuneesti välittää horisontaalisen rotaation, mutta ei kuitenkaan päivitä vertikaalista rotaatiota. Kyseinen toiminnallisuus on moninpelille hyvin tärkeää. Se voi toteuttaa käyttämällä replikoitua muuttujaa esimerkiksi seuraavalla tavalla.

Unrealin pawn-luokassa löytyy valmiiksi replikoitu funktio, mitä voidaan suoraan hyödyntää implementoinnissa.

```
/** Replicated so we can see where remote clients are looking. */
UPROPERTY(replicated)
uint8 RemoteViewPitch;
```

Esimerkkikoodi 15. Esimerkki valmiiksi replikoidusta muuttujasta, joka tallettaa pelaajien katselusuunnan.

Muuttuja on tallennettu ”Uint8”-muotoon, joka käytännössä tarkoittaa sitä, että se on kompressoitu yhden bitin kokoiseksi ja sillä ei ole tässä muodossa negatiivisia arvoja.

```
void APawn::SetRemoteViewPitch(float NewRemoteViewPitch)
{
            // Compress pitch to 1 byte
            NewRemoteViewPitch = FRotator::ClampAxis(NewRemoteViewPitch);
            RemoteViewPitch = (uint8)(NewRemoteViewPitch * 255.f/360.f);
}
```

Esimerkkikoodi 16. Esimerkki miten muuttuja on kompressoitu yhden bitin kokoiseksi Pawn.cpp tiedoston sisällä.

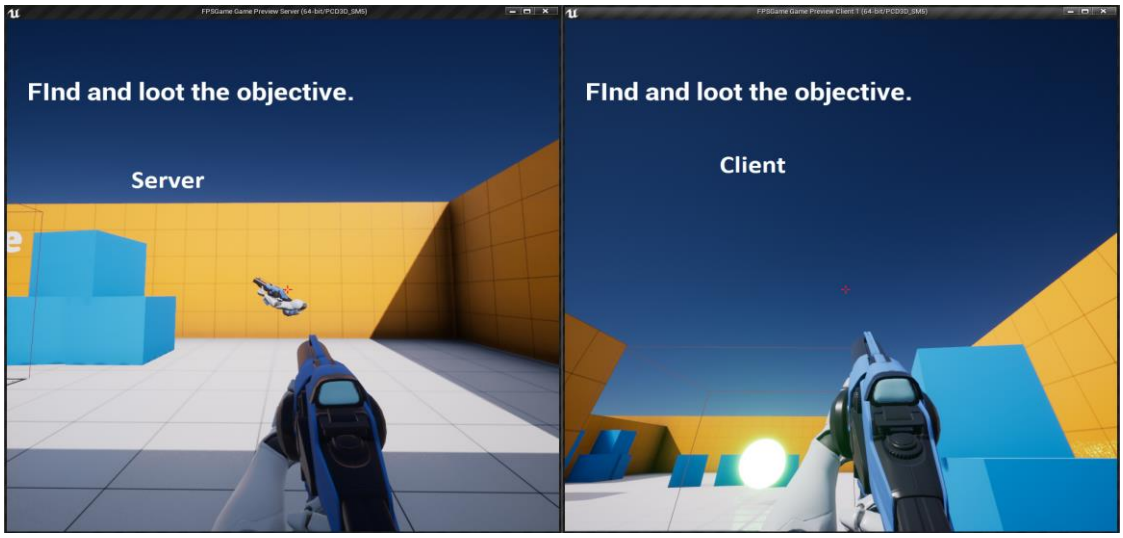
Jotta muuttujaa voidaan hyödyntää oikein toteutuksessa, muuttujaa täytyy dekompressoida takaisin float-muuttujan muotoon. Aloitetaan toiminnallisuuden toteutus luomalla tick-funktio hahmo-luokan header -tiedostossa. Implementoidaan hahmo-luokan cpp -tiedoston tick-funktio ja kirjoitetaan seuraavat koodirivit.

```
void AFPSCharacter::Tick(float Deltatime)
{
Super::Tick(Deltatime);
//tarkistetaan onko kyseinen hahmo paikallisesti ohjattava.
//Kun todetaan että pelaaja ei ole paikallisesti ohjattava, hypätään ehdon si-
sälle
  if (!IsLocallyControlled())
  {
   //Luodaan uusi FRotator -muuttuja, asetetaan
   //sen rotaatio arvoksi kyseisen Actorin rotaatio.
   FRotator NewRot = GetActorRotation();
   if (NewRot.Pitch == 0)
   {
    //Asetetaan uudeksi FRotator -muuttuja
    //dekompressoidun "RemoteViewPitch" -muuttujan arvoksi
    NewRot.Pitch = RemoteViewPitch * 360.0f / 255.0f;
    //Rajoitetaan minimi ja maksimi arvot varmuuden vuoksi uudestaan.
    FRotator::ClampAxis(NewRot.Pitch);
    //Asetetaan kamera komponentin rotaatio arvoksi uusi rotaatio arvo "NewRot".
    CameraComponent->SetWorldRotation(NewRot);
   }
  }
}
```

Esimerkkikoodi 17. Hahmon rotaation replikaation toteutusesimerkki.

Esimerkissä tarkistetaan ensin, onko hahmo paikallisesti ohjattava, sillä haluamme replikoida vain niiden hahmojen rotaatiot, jotka eivät ole kyseisen pelaajan ohjaamia. Hahmolle luodaan uusi muuttuja, jolle asetetaan alkuarvoksi hahmon actor-rotaatio eli ”ActorRotation”. Uuden luodun rotaatio arvon kaltevuudeksi asetetaan dekompressoitu ”RemoteViewPitch”-arvo. Rajoitetaan sen minimi- ja maksimiarvot ja asetetaan se

kamerakomponentin uudeksi rotaatioarvoksi. Tämän jälkeen rotaation pitäisi välittyä muille pelaajille seuraavan kuvan mukaisella tavalla.

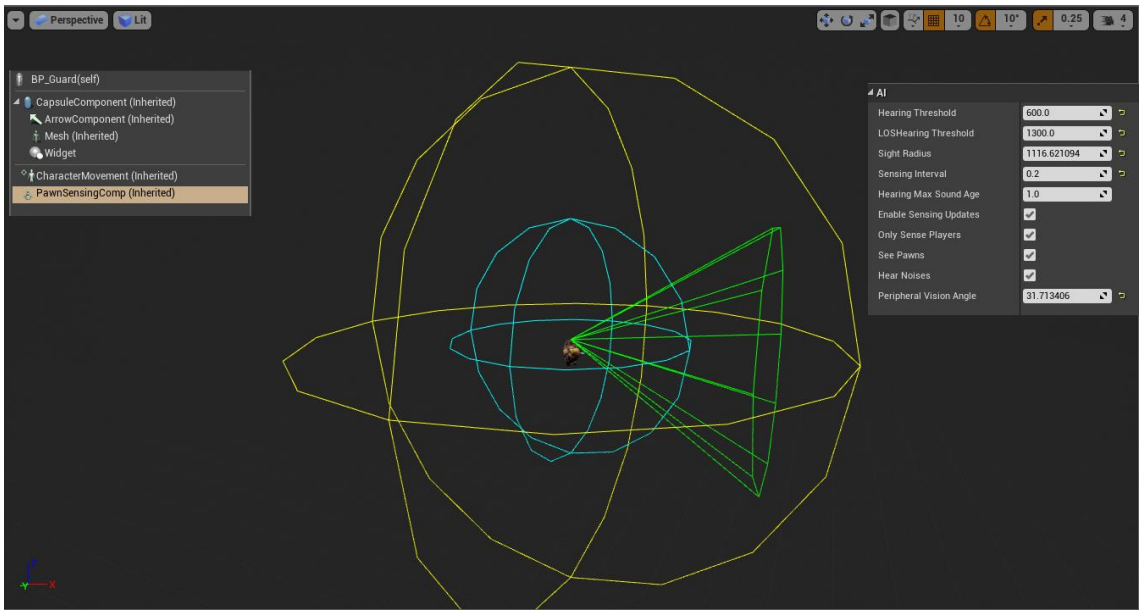


Kuva 30. Onnistuneen tähtäämiskaltevuuden replikaation lopputulos.

Toinen esimerkki liittyy projektin yksinkertaiseen tekoälyn toteutukseen, miten tekoäly vaihtaa tilansa pelinaikana ja miten se välittyy kaikille pelaajille. Toteutuksessa käytetään jälleen muuttujareplikaatiota hyödyntäen ”Rep_Notify”-muuttujaa. Tekoälylle on määritelty eri tiloja seuraavalla tavalla:

```
enum class EAIState : uint8
{
            Idle,
            Suspicious,
            Alerted
};
```

Esimerkkikoodi 18. Esimerkki miten tekoälylle on luotu eri tilat enumeraator-luokan avulla.



Kuva 31. Kuvaesimerkki, miten tekoäly havaitsee pelimaailmassa tapahtuvia asioita.

Kuvan 31 mukaan voidaan nähdä, miten tekoäly havaitsee pelimaailmassa tapahtuvia asioita. Kyseisessä pelissä tekoäly käyttää kuulo- ja näköaistia muuttaakseen oman tilansa (”EAIState”). Kun pelaaja astuu näkökenttään, enumeraator-arvo muuttuu ”alerted” -tilaan ja pelaaja häviää pelin. Pelaajan ampuessa luodin tai kävellessä tekoälyn kuulo alueen sisällä, sen tila muuttuu ”suspicious”-tilaan ja se koittaa katsoa, mitä siellä suunnassa on kolmeksi sekunniksi.

Tekoälyn tila saadaan moninpeli kelpoinen asettamalla muuttuja replokoiduksi ”Rep_notify”-ominaisuudella seuraavalla tavalla.

```
UPROPERTY(ReplicatedUsing=OnRep_GuardState)
EAIState GuardState;

UFUNCTION()
void OnRep_GuardState();
```

Esimerkkikoodi 19. Esimerkki tekoälyluokan header-tiedostossa olevan muuttujan asettamista moninpelikelpoiseksi ”Rep_Notify” ominaisuudella.

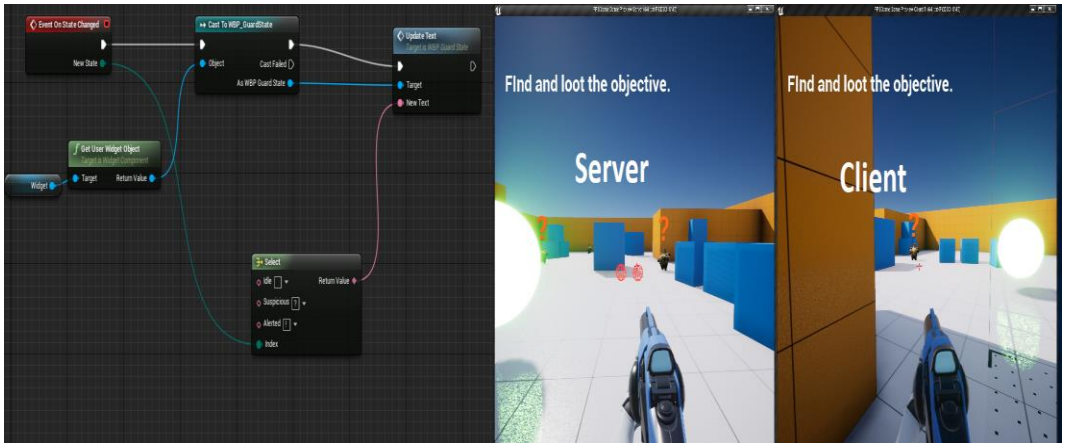
Esimerkkikoodissa nähdään, miten ”Guardstate”-muuttuja asetetaan käyttämään ”OnRep_GuardState” -funktiota replikaatio ominaisuutena. Näin ollen aina kun muuttuja päivittyy, se päivittyy sillä hetkellä muillekin pelaajille. Seuraavassa koodiesimerkissä nähdään, miten funktiot on muutettu tekoälyn cpp-tiedostossa.

```
void AFPSAIGuard::OnRep_GuardState()
{
		OnStateChanged(GuardState);
}

void AFPSAIGuard::SetGuardState(EAIState NewState)
{
		if (GuardState == NewState)
		{
			return;
		}
		GuardState = NewState;
		OnRep_GuardState();
		//no longer needed when using on rep variables
		//OnStateChanged(GuardState);
}
```

Esimerkkikoodi 20. Esimerkki tehdyistä muutoksista tekoälyn cpp-tiedostoon.

Aina kun ”SetGuardState”-funktio kutsutaan, mikä muuttaa tekoälyn tilaa, niin ”OnRep_GuardState” -funktio kutsutaan ja välitetään blueprint toteutettavalle funktiolle tieto, missä tilassa ollaan. Blueprintissä määritetään tekoälylle GUI-elementtejä sen kyseisen tilan perusteella seuraavalla tavalla.



Kuva 32. Kuvaesimerkki, miten tekoäly toimii pelissä.

Kuvassa 32 käy ilmi, miten blueprintin implementoitava funktio on toteutettu tuottamaan visuaalista havaintoa tekoälyn tilasta. Kun tekoäly epäilee kuulleensa jotakin, sen pään päälle asetetaan kysymysmerkki, joka replikoituu nyt onnistuneesti molemmille pelaajille.

6.2 *”RisingCut”*-pelin toteutuksia blueprintillä

”RisingCut” on peliprojekti, joka sai alkunsa koulun peliprojektikurssin aikana. Tavoitteena oli toteuttaa kaksintaistelumiekkailupeli vanhojen tappelupelien hengessä, jossa pelaajat voivat pelata paikallisesti vastakkain samalla koneella, verkon välityksellä, sekä tarvittaessa tekoälyä vastaan. Suurena inspiraationa projektiin oli ps1-konsolille julkaistu ”Bushidoblade” (1997) -tappelupeli, jossa hyödynnettiin yhdenosuman kuolema -periaatetta (”One hit K.O”). Halusimme ottaa asioita joita koimme toimivaksi niissä peleissä ja luoda samalla peli nykymaailmaan soveltuvaksi. Projekti oli myös minun ensi kosketukseni Unreal 4 -pelimoottoriin, joten suurin osa omasta osaamisestani tuli opittua tämän projektin parissa. Projektissa on useita eri moninpeliesimerkkejä, mistä riittäisi kertomista monen sivun verran. Näin ollen yritän ottaa minulle haastavimmat tai minusta mielenkiintoisemmat esimerkit esiin tässä osiossa.

6.2.1 Jaetun kameran Implementointi

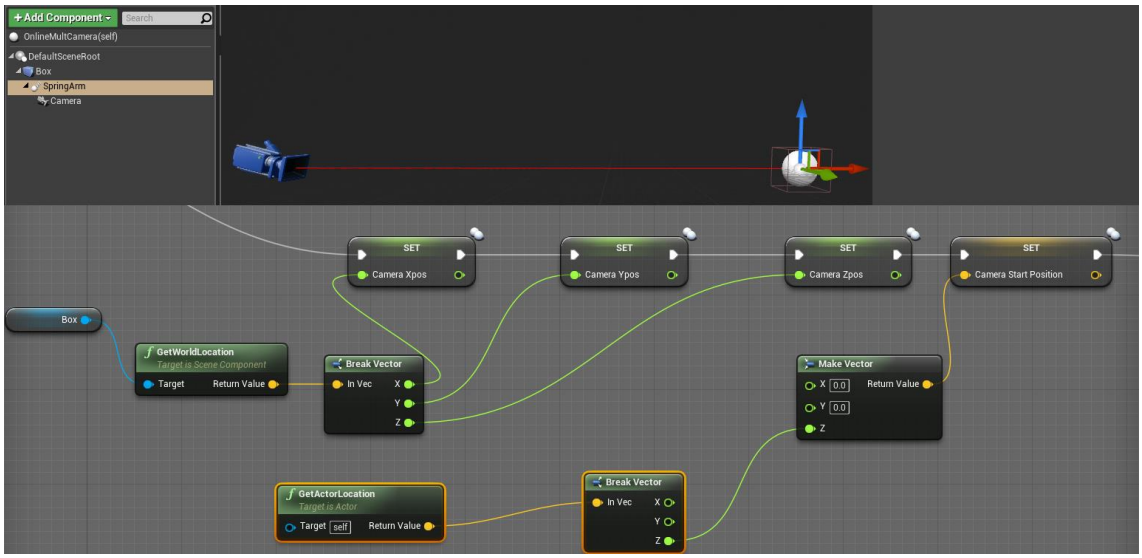
Ensimmäisessä esimerkissä käyn pelissä toimivan jaetun kameran toiminnallisuudet ja haasteita pääpiirteittäin läpi. Seuraavassa kuva esimerkissä nähdään, miten kamera toimii.

Kuva 33. Kuvaesimerkki miten jaettu kamera toimii palvelimella sekä asiakkaan ruudulla samalla tavalla.

Kuvassa 33 nähdään, miten jaettu kamera toimii. Kyseinen kamera oli ehkä hankalimpia asioita saada toimimaan moninpeliympäristössä. Ideana on, että kamera osaa laskea molempien pelaajien sijainnin perusteella oman sijaintinsa kuvakulman pelaajia kohti.

Kameraa piti päivittää näitä tietoja jokaisella ruutupäivityksellä ja tämän piti saada yhdenmukaisesti toimimaan ilman bugeja. Kameran toiminnallisen vaatimuksen takia se oli hyvin herkkä monenlaisille koodillisille virheille, ja yksikin virhe aiheuttaisi pelin rikkoutumisen.

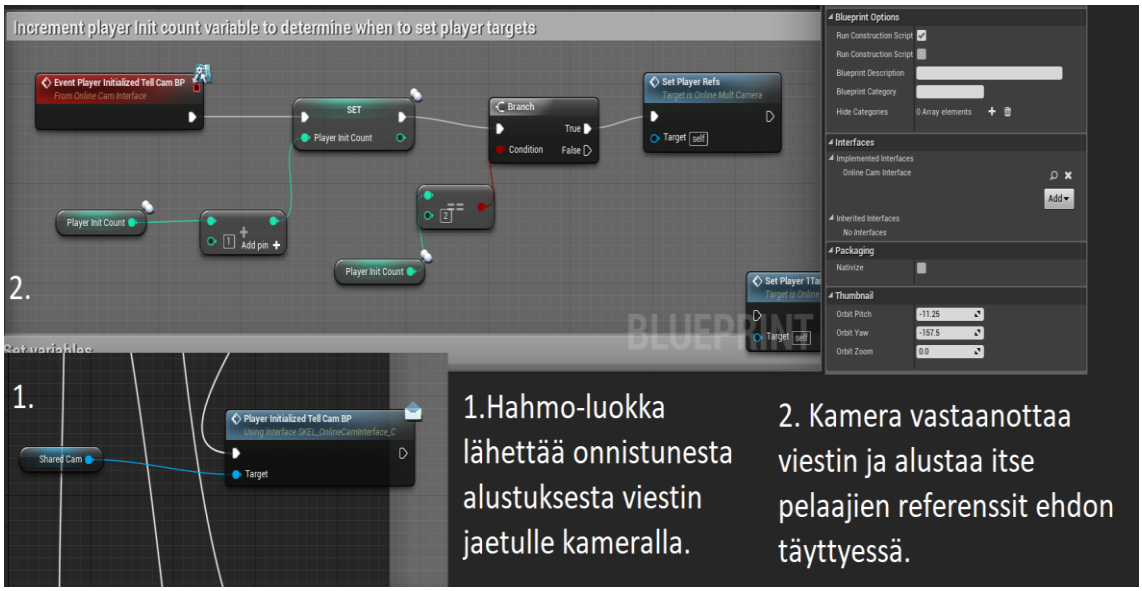
Ensimmäinen vaihe toteutuksessa oli luoda tyhjä actor blueprint -luokka, jossa oli yksinkertainen osumalaatikko (”BoxCollider”), jonka hierarkian alla on kamerapuomi (”SpringArm”) ja sen alla itse kamera. Blueprint-luokka laitettiin replikoitumaan aiemmin mainitun actor-replikaatio menetelmällä, jotta se soveltuisi moninpeliin. Ajatuksena oli hakea molemmista pelaajista sijaintireferenssit ja niiden suhteiden perusteella liikutella osumalaatikon sijaintia ja rotaatiota. Seuraava kuva havainnollistaa, miltä blueprint-luokka näyttää ja miten sijainnit alustetaan luokan luomisen yhteydessä.



Kuva 34. Kuvaesimerkki havainnollistaa luokan sisältöä ja miten sen osumalaatikon sijaintia alustetaan luokan luomisen yhteydessä (”Event begin play”).

Seuraavassa vaiheessa tarvittiin erinäisiä vektorilaskuja, jossa määriteltiin pelaajien välinen keskipiste, niiden maksimietäisyys toisistaan, pelaajien suunta vektorien laskenta rotaatioita varten ja actorin liike ja -sijaintien päivittämisfunktioiden toteuttaminen. Näitä laskuja varten tarvittaisiin pelaajien referenssit, jotka aluksi vaikuttivat yksinkertaisilta saada, mutta aiheuttivat suurta hämmennystä ja tuskailua toteuttamisen aikana. Suurin ongelma oli, että pelaajien alustusaika tai järjestys ei ollut aina sama. Tämä aiheutti sen, että jommankumman pelaajien referensseistä olisi epäpätevä hyvin harvinaisissa tilanteissa. Koodin ajaminen oli myös riippuvainen siitä, milloin molemmat pelaajat olivat yhdistyneet palvelimelle, joka myös aiheuttaisi vastaavanlaista ongelmaa. Lisäksi pelaajien

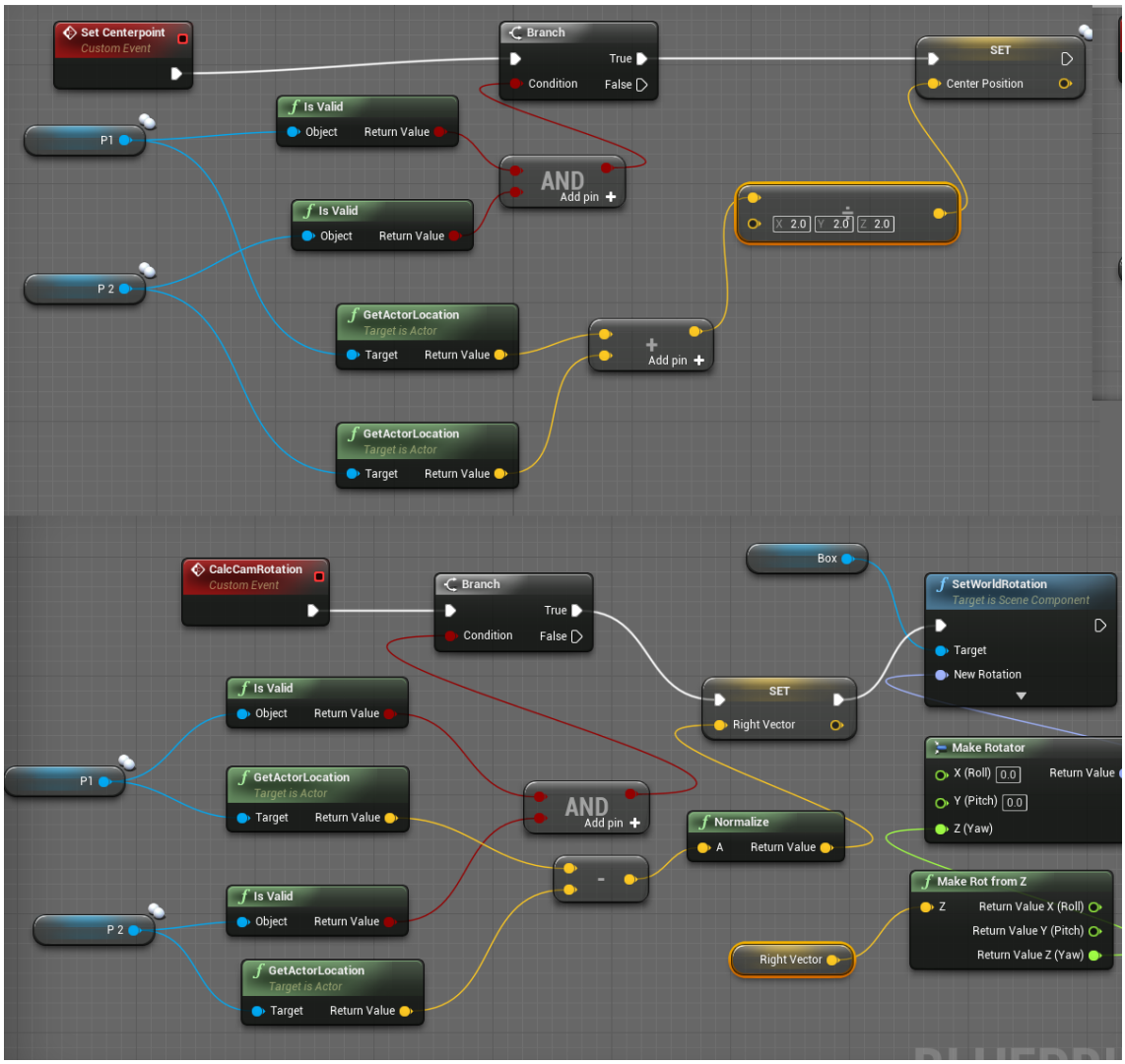
referenssit pitäisi uusia aina, kun pelaaja kuolee ja luodaan uudestaan. Lopulta kuitenkin onnistuin hyödyntämään pelimoottorissa olevaa ”blueprint interface” -menetelmää, joka käytännössä lähettää viestin kaikille sille tilaaville luokille ja kertoo heille, että voivat suorittaa interfacen sisällä olevaa tapahtumaa ja ajaa koodia sen perusteella. Seuraava kuvaesimerkki havainnollistaa inferfacen käyttöä peliprojektissa.



Kuva 35. Kuvaesimerkki interface-menetelmän toiminnallisuudesta.

Tässä tilanteessa halutaan, että hahmoluokka lähettää viestin onnistuneesta alustuksesta jaetulle kameraluokalle. Kameran saatua viestin hahmoluokalta kamera lisää sen muuttujaan. Kun kameraluokka toteaa, että kaksi pelaajaa on alustettu, luokassa ajetaan pelaajien referenssien alustus koodi. Tämän toteutuksen avulla saadaan aina pätevä referenssi molempiin pelaajiin riippumatta hahmoluokan alustus ajasta tai järjestyksestä.

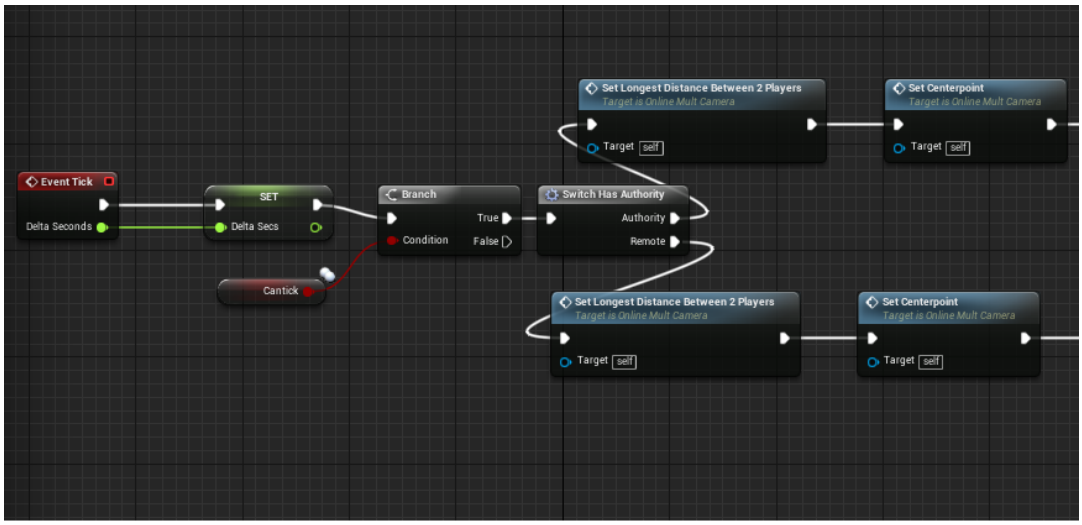
Tämän jälkeen voidaan onnistuneesti hyödyntää ja laskea vektorilaskelmia pelaajien sijainneista ja rotaatioista. Seuraavassa kuvassa näkyy, miten keskipisteen ja rotaation laskennat on toteutettu.



Kuva 36. Kuva esimerkki, miten kameran keskipiste ja rotaatio toteutetaan.

Molemmissa esimerkeissä tarkistetaan ensin, onko molempien pelaajien referenssit olemassa olevia ”Is Valid” -tarkistuksella. Tämän jälkeen suoritetaan tarvittavat vektorilaskut. Keskipisteen laskennassa lasketaan sijaintien välinen vektorin pituus, joka jaetaan kahdella ja saadaan tarvittava keskipiste. Kameran rotaatiossa haetaan hahmojen sijaintien perusteella normalisoitua vektoria, mitä jaetun kameran osumalaatikko voi hyödyntää oman rotaation päivittämiseen.

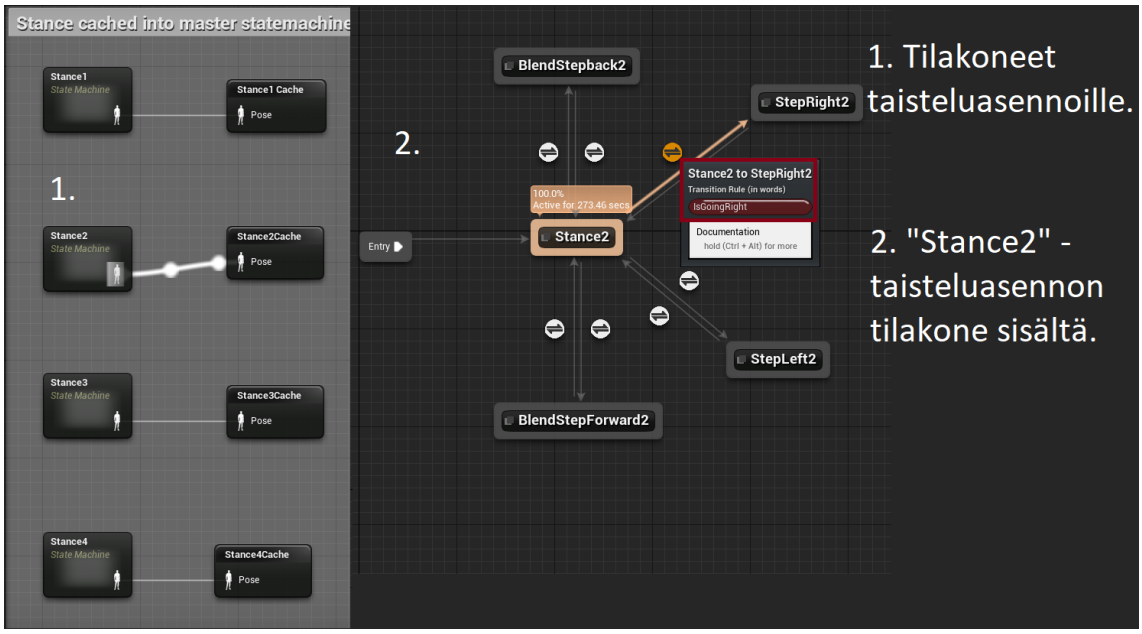
Toteutukset kutsutaan muiden funktioiden kanssa jokaisella ruutupäivityksellä hyödyntäen ”Switch has authority” -makroa, joka päivittää samat arvot kameraluokan asiakkaalle ja palvelimen puolelle. Seuraavassa kuvassa voidaan havainnollistaa ruutupäivityksen implementointia jaetun kameran luokassa.



Kuva 37. Kuvaesimerkki jaetun kameran funktiokutsuista jokaisella ruutupäivityksellä.

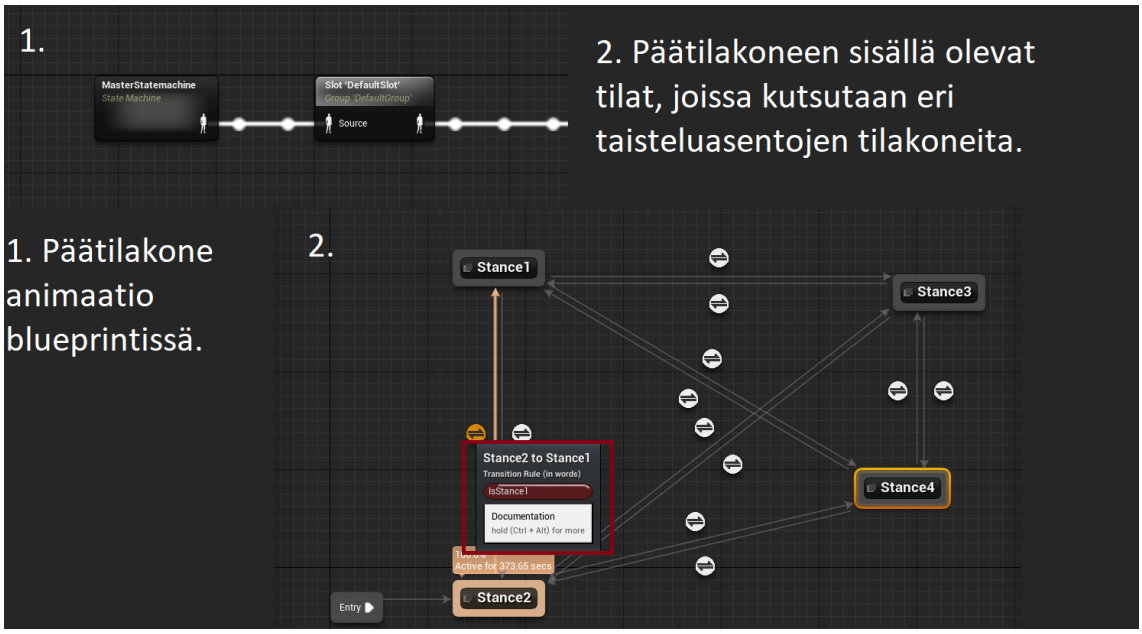
6.2.2 Animaatioiden toteutus ja replikointi

Seuraava esimerkki on myös hyvin olennainen osa ”RisingCut”-peliä, sillä peli on hyvin riippuvainen animaatioista. Animaatioiden replikointi on suhteellisen yksinkertaista pääasiassa, mutta kun projektissa tarvitaan useita eri animaatioita ja halutaan ne moninpelikelpoiseksi, toteutukset voivat muuttua melko monimutkaiseksi. Peli suunniteltiin niin, että jokaisella hahmolla olisi neljä eri taisteluasentoa, jossa kaikissa olisi kaksi lyönti iskua, torjuntaisku, kuolema-animaatio sekä tyrmäysanimaatio. Jokaiselle taisteluasennolle piti olla omat kävelyanimaatiot kahdeksaan eri suuntaan, eli oikealle vasemmalle, eteenpäin, taaksepäin ja viistosuuntiin. Animaatioiden määrä nousi yli sadan jokaista hahmoa kohden, joten hahmoille piti kehittää animaatio systeemi, joka vaihtaisi sulavasti animaatiosta toiseen. Ratkaisuksi toteutin jokaisen hahmon taisteluasennon animaatioille oman tilakoneen. Tilakoneet luodaan Unrealissa olevan animaatio-blueprintin animaatiograafin käyttöliittymässä (”AnimGraph”). Jokaisen taisteluasennon tilakoneella oli omat kävelyanimaatiot. Kaikki taisteluasennot välitettiin ”Master”-tilakoneeseen, joka huolehti siitä mitä taisteluasentoa kuuluisi käyttää. Seuraavassa kuvaesimerkissä voi paremmin hahmottaa miltä se näyttää.



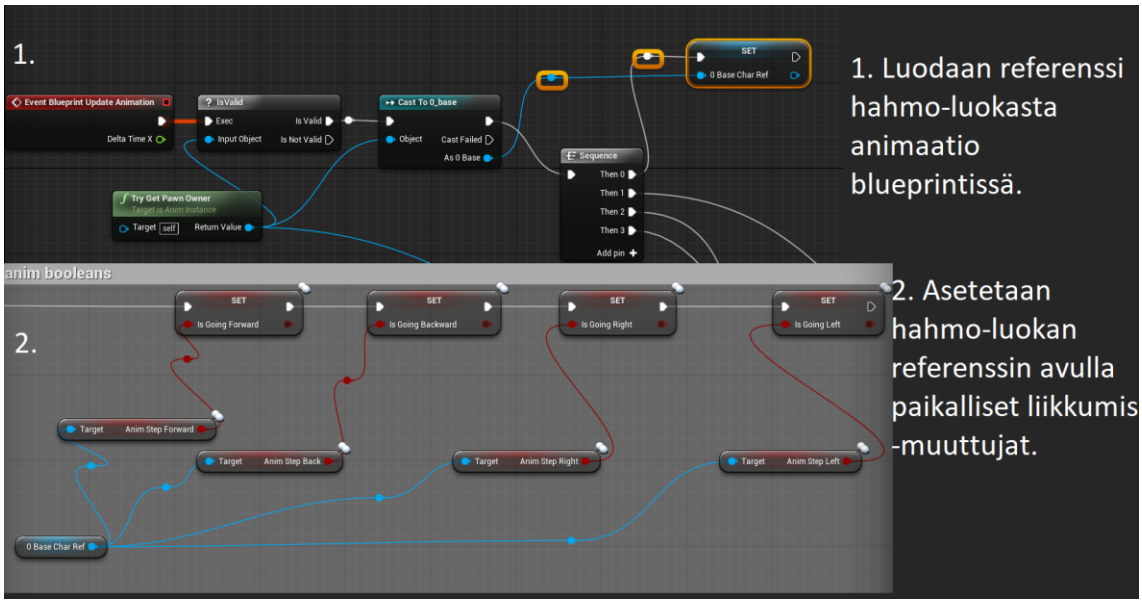
Kuva 38. Kuvaesimerkki tilakoneen toteutuksesta.

Kuvassa 38 nähdään, miten tilakone on toteutettu animaatio blueprintin animaatiokäyttöliittymässä. ”Stance”-taisteluasennon ollessa aktiivinen tilakoneelle määritetään ehto, joilla eri animaatiotilat (”state”) aktivoidaan. Esimerkiksi jos muuttuja ”IsGoingRight” on aktiivinen, animaatioille aktivoidaan oikealle kävelevä animaatio. Jokainen taisteluasennon tila tallennetaan välimuistiin (”CachePose”) ja välitetään animaatio blueprintin päätilakoneelle (”master statemachine”). Päätilakone määrittelee minkä taisteluasennon tilakone tulisi olla päällä pelaajan käskyjen perusteella, seuraavasta kuvasta 39 käy ilmi, miten tämä toimii.



Kuva 39. Kuvaesimerkki päätilakoneesta animaatio-blueprintissä.

Kuvassa 39 voidaan havainnoida, miten päätilakone ohjaa eri taisteluasentojen tilakoneita. Päätilakone tarkistaa, onko muuttuja ”IsStance1” tosi. Jos se on päätilakone, asettaa ”Stance1” tilakoneen aktiiviseksi. Tilakoneita aktivoidaan boolean muuttujilla, jotka muuttuvat pelaajien käskyjen mukaisesti. Ennen kun muuttujia voidaan hyödyntää, niiden ohjattavien hahmojen tiedot täytyy välittää kyseiselle animaatio blueprintille. Seuraava kuva esimerkki esittää, miten hahmoista saadaan tarvittavat tiedot ja miten tieto asetetaan animaatioiden käytettäväksi.



Kuva 40. Kuvaesimerkki, miten välitetään hahmoluokan tiedot animaatio-blueprintin käytettäväksi.

Kuvasta 40 käy ilmi, miten tiedot välitetään animaatioiden aktivoimiseen. Ensin käytetään ”Cast To” -solmua, jotta voidaan saada referenssi asetettua hahmoluokasta. Tämän jälkeen haetun referenssin avulla voidaan hakea luokan sisällä olevia muuttujia ja niillä asettaa animaatio-blueprintissä olevia paikallisia muuttujia. Samalla periaatteella voidaan hakea esimerkiksi pelaajan kävelynopeutta ja säädellä animaatioiden nopeuksia tai aktivoida hyökkäysanimaatiota. Jotta animaatiot saadaan moninpelikelpoiseksi, hahmoluokan muuttujat täytyvät olla replikoituja. Muuten tieto vaihtuneesta animaatiosta ei välity palvelimelle tai muille pelaajille.

7 Yhteenveto

7.1 Omien kokemusten kompastuskivet ja opit

Aloitin Unreal 4 -pelimoottorin käytön melko vähäisellä ohjelmointikokemuksella. Minulla oli aiempaa ohjelmointikokemusta koulusta käydyistä kursseista ja olin käyttänyt Unity-pelimoottoria koulun peliprojektissa. Jälkeenpäin katsottuna, kun aloitin *Rising Cut*-pelin kehittämisen, otin vähän liian ison projektin tehtäväksi ensimmäiseksi Unreal 4 -peliksi. Tarkoitus oli, että peli olisi vain paikallisesti pelattava, mutta opetusmateriaalia ei juuri löytynyt, jolla pääsisi alkuun projektin kanssa. Näin päädyin ensin toteuttamaan netissä

pelattavaa version pelistä, sillä siitä löytyi paljon opetusmateriaalia. Olin myös projektin ainoa ohjelmoija, jolla ei ollut paljon kokemusta, joten projektin eteneminen oli täysin minun vastuulla. Olisi ollut parempi idea ottaa pienempiä projekteja aluksi, joilla olisi päässyt opettelemaan paremmin pelimoottorin rakennetta ja toimintatapoja. Moninpelin toteuttaminen ei ollut myöskään kovin fiksu valinta ensimmäiseksi projektiksi. Koen kuitenkin, että sinnikkyydellä ja kovalla työllä onnistuin saamaan asioita toimimaan, vaikka niihin meni paljon enemmän aikaa. Olisin halunnut aikaisemmin siirtyä käyttämään Unrealissa C++ -kieltä, jotta voisin soveltaa paremmin koulussa opittuja taitoja. Koin, että minun oli pakko keskittyä vain pelin toteutuksien implementointiin, kun olin ainoa ohjelmoija kyseisessä projektissa. Toisaalta ”Rising Cut” -pelissä ei ollut väliä, käyttääkö blueprint-skriptausta, sillä pelissä ei ole kuin kaksi hahmoa samaan aikaan luotuna, vaikka C++ -kielen käyttäminen olisi ollut minulle hyödyllisempää. Pelissä on myös paljon asioita mitkä pitäisi siivota ja toteuttaa uusiksi asioiden luettavuuden selkeyttämiseksi.

7.2 Insinöörityön yhteenveto

Unreal 4 -pelimoottorilla on mahdollista toteuttaa moninpelejä melko vähäisellä ohjelmointikokemuksella ja koen, että se on mainio tapa oppia ohjelmointilogiikkaa ja pelien kehittämistä. Blueprintin skriptauskieli on hyvin selkeä ja helppo tapa ohjelmoida peliprototyyppejä tai pieniä kokonaisia pelejä. Kielen logiikka on myös melko yhtenäinen Unreal 4 C++ -kielen kanssa. Se, että moninpelejä pystyy toteuttamaan blueprinttien avulla pelkästään pelimoottorin tarjoamilla työkaluilla, on erittäin hyödyllistä ja se on toteutettu käyttäjille yksinkertaiseksi. Pelimoottorin ympärillä on muodostunut aktiivinen yhteisö, jotka mielellään auttavat uusia kehittäjiä Unreal 4 -sivuston ”answerhub” kyselyfoorumin kautta. Kokeneemmat freelancerkehittäjät ovat myös ottaneet roolin tuottamalla ilmaisia tutoriaaleja ja opetusmateriaalia pelimoottorin eri osioista.

Vaikka Unreal 4 -pelimoottorin tarjoamat työkalut ovat intuitiivisia ja helppokäyttöisiä, pelimoottorissa on myös puutteita. Dokumentaatiot ovat pääasiassa hyvät, mutta suhteellisen yksinkertaiset, ja useimmiten ne ovat monta versiota jäljessä. Moninpeleille Unreal 4 soveltuu pääasiassa tosi hyvin, mutta aikaisemmin se ei tukenut hirveen hyvin uusia suosittuja ”Battle Royale” -tyyppisiä pelejä, joissa on sata pelaajaa samaan aikaan isolla kartalla. Ongelma on siinä, että pelimoottoria ei ole alun perin suunniteltu toimimaan niin isolla pelaaja määrällä. Isot pelaajamäärät ja kentät tarkoittavat sitä, että datan liikkumisen ja piirtokutsujen määrä on hyvin suuri, mikä hidastaa pelipalvelimia. Palvelimien

päivitysnopeudet, jotka ovat valmiiksi melko alhaiset, kärsivät kyseisten pelien alkutilanteissa, aiheuttaen hyvin epäjohdonmukaisen pelikokemuksen pelaajille. Toisaalta, Unrealin kehittäjät (”Epic games”), ovat huomanneet tarpeen parantaa nettikoodiaan tukemaan ”Battle Royal”-peligenren pelejä. Jopa heidän kehittämänsä peli ”Fortnite” kuuluu samaan peligenreen. Epic Games piti luennon Unreal-pelimoottorin uusimmasta päivityksestä kevään GDC 2018 -tapahtumassa, (”Game Developers Conference”) johon sisältyi useita optimointeja ”Battleroyale” -peleihin, erityisesti heidän kehittämälleen ”Fornite”-pelilleen. Parannuksiin liittyy esimerkiksi pelimoottorin pääsäikeeseen (”Game thread”), kuten animaatioiden, hahmojen liikkumisen ja replikoinnin merkillisyyden optimointeja. Lisää aiheesta voi katsoa seuraavista lähteen YouTube -linkeistä, joissa kehittäjät käyvät läpi yksityiskohtaisesti parannuksia. (22; 23.)

Onnistuin pääasiassa pääsemään asettamiini tavoitteisiin tässä työssä. Sain laajemman taustatiedon moninpeliin liittyvistä haasteista ja niiden yleisimmistä ratkaisutavoista. Esimerkiksi, miten pelipalvelimen päivitys- ja simulaationopeudet voivat vaikuttaa pelin suoritukseen ja miten viivettä voidaan kompensoida pelaajan puoleisella ennustuksella. Laajensin myös omaa osaamistani Unreal 4 -pelimoottorin suhteen. Opin hyödyntämään omia koulussa opittuja C++ -kielen taitoja pelimoottorin kanssa, myös moninpelitoteutuksen näkökulmasta. Olisin toivonut saavani vielä parempia C++ -kielellä tehtyjä esimerkkejä, mutta aikaa oli rajallisesti muiden töiden ohella. Koen kuitenkin työssä olevien esimerkkien olevan helpommin lähestyttäviä kokemattomille kehittäjille ja olen rakentanut itselleni myös pohjan, mistä voin jatkaa omaa oppimista Unreal 4 -pelimoottorin ja pelinkehittämisen parissa.

Lähteet

1 Yleisesti Moninpeleistä. Multiplayer Game Programming: An Overview of NetworkedGames:http://ptgmedia.pearsoncmg.com/images/9780134034300/samplepages/9780134034300.pdf.Luettu 16.03.2018.

2 Yleisesti Moninpeleistä:http://www.informit.com/articles/article.aspx?p=2461064.Luettu 16.03.2018.

3 Moninpeliin liittyvät haasteet.http://trac.bookofhook.com/bookofhook/trac.cgi/wiki/IntroductionToMultiplayerGameProgramming.Luettu 16.03.2018.

4 Viive.https://en.wikipedia.org/wiki/Lag.Luettu 17.03.2018.

5 Battlenonsense youtube videosarja: yleisesti moninpeleistä -ja ratkaisutapoja. https://www.youtube.com/watch?v=hiHP0N-jMx8.Luettu 23.03.2018.

6 Valve peliyhtiön moninpeli toteutukset -ja ratkaisutavat.https://developer.valvesoftware.com/wiki/Latency_Compensating_Methods_in_Client/Server_Ingame_Protocol_Design_and_Optimization.Luettu 23.03.2018.

7 Ekstrapolointi. https://en.wikipedia.org/wiki/Extrapolation.Luettu 23.03.2018.

8 Interpolointi.https://fi.wikipedia.org/wiki/Interpolaatio.Luettu 23.03.2018.

9 Unreal 4 virallinen dokumentaatio. Verkko-osio. https://docs.unrealengine.com/latest/INT/Gameplay/Networking/Blueprints/index.html.Luettu 19.9.2017.

10 Cedric Neukirchen. UE4 Network Compendium. http://cedric-neukirchen.net/Downloads/Compendium/UE4_Network_Compendium_by_Cedric_eXi_Neukirchen.pdf.Luettu 23.03.2018.

11 Unreal 4 virallinen dokumentaatio. Blueprints-osio. https://docs.unrealengine.com/latest/INT/Engine/Blueprints/GettingStarted/index.html/.Luettu 19.9.2017.

12 Unreal 4 tutoriaalit. Network tutorial series. https://www.unrealengine.com/en-US/blog/blueprint-networking-tutorials.Luettu 17.9.2017.

13 Unreal 4 virallinen dokumentaatio. Gameplay Framework-osio. https://docs.unrealengine.com/latest/INT/Gameplay/Framework/index.html.Luettu 17.9.2017.

14 Unreal 4 virallinen dokumentaatio. RPC-osio. https://docs.unrealengine.com/latest/INT/Gameplay/Networking/Actors/RPCs/index.html/.Luettu 18.9.2017.

15 Unreal 4 virallinen dokumentaatio. Collapsing graphs. https://docs.unrealengine.com/latest/INT/Engine/Blueprints/BP_HowTo/CollapsingGraphs/.Luettu 20.9.2017.

16 Unreal 4 virallinen dokumentaatio. Events. https://docs.unrealengine.com/latest/INT/Engine/Blueprints/UserGuide/Events/.Luettu 21.9.2017.

17 Unreal 4 virallinen dokumentaatio. Gameplay Framework overview. https://docs.unrealengine.com/latest/INT/Gameplay/Framework/QuickReference/index.html.Luettu 23.9.2017.

18 Unreal 4 virallinen dokumentaatio. Blueprint overview. https://docs.unrealengine.com/latest/INT/Engine/Blueprints/Overview/index.html/.Luettu 22.9.2017.

19 Unreal 4 virallinen dokumentaatio. Blueprint FAQ and Tips. https://wiki.unrealengine.com/Blueprint_FAQ_and_Tips.Luettu 28.3.2018.

20 Unreal 4 virallinen dokumentaatio, Programming introduction. https://docs.unrealengine.com/en-us/Programming/Introduction.Luettu 24.03.2018.

21 Unreal 4 Wiki Replication. https://wiki.unrealengine.com/Replication.Luettu 24.03.2018

22 Optimizing UE4 for Fortnite:Battle Royale part1 GDC 2018 Unreal Engine. https://www.youtube.com/watch?v=KHWquMYtji0.Luettu 2.4.2018.

23 Optimizing UE4 for Fortnite:Battle Royale part2 GDC 2018 Unreal Engine. https://www.youtube.com/watch?v=1xiwJukvb60.Luettu 2.4.2018.

24 Gabriel Gambetta Client-side Prediction live demo. http://www.gabrielgambetta.com/client-side-prediction-server-reconciliation.html.Luettu 12.4.2018.