jamk

Quantum, deterministisen pelimoottorin tuomat hyödyt ja tehokkuudet pelinkehityksessä

Deterministinen moninpeli-pelimoottori Unityssä

Henri Mäkinen

Opinnäytetyö, AMK
Kesäkuu 2024
Tieto- ja viestintätekniikan tutkinto-ohjelma

jamk | Jyväskylän ammattikorkeakoulu
University of Applied Sciences

jamk

Kuvailulehti

Mäkinen, Henri

Quantum, deterministisen pelimoottorin tuomat hyödyt ja tehokkuudet pelinkehityksessä
Deterministinen moninpeli-pelimoottori Unityssä

Jyväskylä: Jyväskylän ammattikorkeakoulu. **Kesäkuu 2024**, 48 sivua.

Tieto- ja viestintätekniikan tutkinto-ohjelma. Opinnäytetyö AMK.

Julkaisun kieli: suomi

Julkaisulupa avoimessa verkossa: kyllä

Tiivistelmä

Pelialan verkkomoninpelien kehittäminen on aina ollut haastavampaa kuin yksinpelien kehittäminen. Tästä syystä verkkomoninpelien kehitystä on yritetty helpottaa, yksinkertaistaa ja tehostaa ensimmäisten verkkomoninpelien ilmestymisestä asti. Nykypäivän verkkomoninpelien kehityksessä hyödynnetään erilaisia pelimoottorin deterministisyyteen pohjautuvia, ohjelmallisia kooditekniikoita korjaamaan verkon yli pelaamisesta aiheutuvia verkko-ongelmia. Pelimoottorin riittävä deterministisyys varmistaa yhdistäneiden pelaajien responsiivisen ja sulavan pelikokemuksen, mutta pelinkehittäjien pitää käyttää täysin determinististä pelimoottoria tai ylläpitää deterministisyyttä ohjelmoidessa epädeterministisellä pelimoottorilla.

Unity Technologies tarjoaa Unity-pelimoottoria, joka on laajasti käytössä oleva epädeterministinen pelimoottori. Unity tarjoaa omia moninpelikehityspaketteja kehittäjien käyttöön käytettäväksi Unity-pelimoottorin päällä, jotka ovat osaksi deterministisiä ja pyrkivät deterministisyyteen verkkomoninpelitoiminnallisuuksia käsitellessä. Photon Engine tarjoaa erilaisia verkkomoninpelikehityspaketteja käytettäväksi Unity-pelimoottorin päällä, joista Quantum-kehityspaketti on täysin deterministinen.

Työn tavoitteena oli selvittää täysin deterministisen pelimoottorin tuomat hyödyt ja tehokkuudet pelinkehityksessä kahden Unity-prototyypin toteutuksen kautta. Ensimmäisen prototyypin toteutus tehtiin osaksi deterministisellä Unity Netcode for Entities-paketilla, kun taas toisen prototyypin toteutus tehtiin täysin deterministisellä Unity Photon Quantum-paketilla.

Opinnäytetyön tuloksina saatiin realistinen kuva verkkomoninpelien deterministisyyden ylläpidosta ja toteutuksien työnkulusta Entities- ja Quantum-paketeilla. Tämän lisäksi ymmärrettiin pakettien verkkomoninpelikehityksen toimintoja ja vahvuuksia.

Avainsanat (asiasanat)

Unity, Photon, pelinkehitys, moninpeli, pelimoottori

Muut tiedot (salassa pidettävät liitteet)

-

jamk

Description

Mäkinen, Henri

Quantum, Benefits and efficiencies of deterministic game engine in game development
Deterministic multiplayer game engine in Unity

Jyväskylä: JAMK University of Applied Sciences, June 2024, 48 pages.

Degree Programme in Information and Communication Technology. Bachelor´s thesis.

Permission for open access publication: Yes

Language of publication: Finnish

Abstract

The development of online multiplayer games has always been more challenging than making single-player games in the gaming industry. For this reason, attempts have been made to make multiplayer game development easier, simpler and more efficient from the release of first multiplayer games. In the development of today's multiplayer games, there are various programmatic code techniques in use to improve the network problems of multiplayer games, which are based on the determinism of the game engine. Sufficient determinism in game engine ensures responsive and smooth gaming experience for connected players, but the game developers need to use fully deterministic game engine or upkeep determinism while coding at nondeterministic game engine.

Unity Technologies offers the Unity game engine, which is a widely used non-deterministic game engine. Unity offers its own multiplayer development packages for developers to use in Unity engine, which are partly deterministic and strive for determinism when handling online multiplayer functionalities. Photon Engine offers a variety of online multiplayer development packages to use in the Unity game engine, from which Quantum development package is fully deterministic,.

Aim of the work was to find out the benefits and efficiencies of a fully deterministic game engine in game development through two Unity prototype implementations. The implementation of the first prototype was done with partly deterministic Unity Netcode for Entities package, while the other prototype implementation was done with entirely deterministic Unity Photon Quantum package.

From the results of the thesis we got a realistic picture of maintaining determinism and the workflow of implementations with the Entities and Quantum packages. In addition to this, we got understanding of the packages online multiplayer development functions and strengths.

Keywords/tags (subjects)

Unity, Photon, game development, multiplayer, game engine

Miscellaneous (Confidential information)

-

Sisältö

Kuvat

Käsitteet

API = Application Programming Interface on ohjelmisto,
joka prosessoi ja lähettää tietoa sovelluksesta toiseen.

Avoin lähdekoodi = ohjelmisto, jossa tekijänoikeudet sallivat käyttäjien käyttää,
muuttaa ja jakaa ohjelmistoa kaikkiin tarkoituksiin.

ECS = Entity Component System on videopeli kehitys ohjelmistoarkkitehtuuri,
jossa kappaleita ei määritellä tyyppien avulla, vaan kappaleet määritellään
entity-kappaleina niissä kiinni olevien komponenttien avulla.

Instanssi = yksittäinen luokan tai kappaleen ilmentymä,
jonka kanssa voi vuorovaikuttaa vaikuttamatta muihin ilmentymiin.

Kirjasto = kokoelma ennalta kirjoitettuja skriptejä ja koodeja, joita voidaan käyttää
eri ohjelmissa kutsumalla kirjaston toiminnallisuuksia itse muokkaamatta kirjastoa.

Legacy = vanhentunut teknologia tai ohjelmisto, jota ei tueta uusissa julkaisuissa tai laitteissa.

Matchmaking = moninpelien järjestelmä, joka yhdistää pelaajat samaan peliin.

Monialusta = ohjelmisto, joka toimii useammalla eri laitteistoalustalla.

MMO-peli = Massive Multiplayer Online-peli on verkkomoninpeli,
jossa suuri määrä pelaajia voi yhdistää samaan pelisessioon.

Netcode = kaikkien verkossa toimivien moninpelien päätelaitteiden välinen
kommunikoimisen koodi, joka vastaa kaikista päätelaitteiden välisistä toiminnoista,
kuten yhdistämisestä, synkronoinnista, datan lähettämisestä ja vastaanottamisesta.

Ohjain = jokaiselle pelaajalle määriteltävä luokka, joka toimii
ohjelmistokäyttöliittymänä pelattavan kappaleen ja pelaajan välissä.

Ohjelmistoarkkitehtuuri = ryhmä ohjelmistorakenteita ja suhteita,
jotka muodostavat ohjelmistojärjestelmän toimintasuunnitelman.

Ohjelmistokehys = ohjelmiston runko, joka tarjoaa erilaisia
toiminnallisuuksia rungon päälle ohjelmoitavaan ohjelmistoon.

Ristiinpelaaminen = moninpelin pelisessioon yhdistäminen
useammalla eri laitteistoalustalla samanaikaisesti.

Scripti = päätelaitteessa reaaliajassa suoritettava koodi,
jota ei käännetä tietokoneen ymmärrettäväksi kieleksi etukäteen.

Skaalautuvuus = ohjelmiston kyky jatkaa toimintaa kasvavan kuorman alla.

Tickrate = pelin päivitysnopeus, eli kuinka usein peli siirtää
tapahtumia ja informaatiota sekunnissa eteenpäin.

Threading = prosessin komponenttien samanaikaisesti suorittamista.

Vasteaika = kokonaisaika, joka menee palvelupyynnön prosessoimisessa ja vastaamisessa.

Verkkohyökkäys = haittamielinen hyökkäys verkon yli,
jolla pyritään tukkimaan, kaatamaan tai saada tietoja palvelusta.

1 Johdanto

Peliteollisuus on ollut nopeasti kasvava ala 1970-luvulta lähtien, niin nopeasti että peliteollisuudessa arvioidaan nykypäivänä liikkuvan yli 100 miljardia euroa vuosittain (Wijman 2018). Tämä arvio tietysti heittää jokaisen eri lähteen mukaan, sillä on mahdotonta mitata tai arvioida jokaisen peliteollisuuden tuottavan osan tuottoja maailmanlaajuisesti. Peliteollisuudessa liikkuvista rahoista tulee nykypäivänä suurin osa kiinan mobiilipeli markkinoista, joka on myös taloudellisesti nopeinten kasvava osa (China Mobile Games Market & 5-Year Forecast Report 2023 2023).

Peliteollisuus on kokenut räjähtävää suosion nousua vielä lähivuosinakin, sillä 2014 vuonna julkaistiin yli kaksi kertaa enemmän videopelejä eniten käytetyssä videopelien jakelualustalla verrattuna edellisvuoteen (Clement 2023). Videopeli ilmiö on selitettävissä ihmisten kehittyneen vapaa-ajan viihteen tarpeen avulla, sillä ihmiset haluavat vuosi vuodelta edistyneempiä ajanviete metodeja. Vuosittainen peli alan kasvu tarkoittaa myös, että ala työllistää joka vuosi enemmän työntekijöitä ja nopea kasvu tuo mukanaan teknologiallisesti nopean etenemisen. Nopea teknologiallinen eteneminen on kuluttajille tietysti erinomainen etu, mutta peli alan kehittäjien tarvitsisi aina olla teknologian kärjessä uusilla työkaluilla nopean ja tehokkaan työnteon turvaamiseksi.

Tämä opinnäytetyö käsittelee yhtä suurta peliteollisuuden teknologia lohkoa, verkkomoninpeli teknologioita. Opinnäytetyössä perehdytään ensin verkkomoninpelien historiaan, palvelinmalleihin, pelimoottoreihin ja verkkomoninpelien ongelmiin, sekä Unity Technologiesin tarjoamiin moninpeli työkaluihin ja Exit Gamesin Photon verkkomoottorin tarjoamiin palveluihin. Tämän jälkeen perehdytään moninpelien käytäntöön toteuttamalla Unity Netcode- ja Photon Quantum-verkkomoninpeli prototyypit. Lopuksi vertaillaan prototyyppien toteutuksien kehittämisprosessia ja verkko-ohjelmointia. Opinnäytetyön lähtökohtana oli perehtyä, toteuttaa ja vertailla kahden eri verkkomoninpeli toteutuksen prosessia ja lopputulosta kokemattoman tekijän pohjalta, jonka tuloksena piti syntyä realistista dataa täysin deterministisen pelimoottorin tuomista hyödyistä moninpeli kehityksessä.

2 Moninpeleistä ja moninpeli-pelimoottoreista

Historiallisesti ensimmäinen kaupallisesti onnistunut samalla päätelaitteella pelattava tietokone moninpeli oli Pong (1972), jonka kehittämisen jälkeen arcade-tyyliset saman päätelaitteen

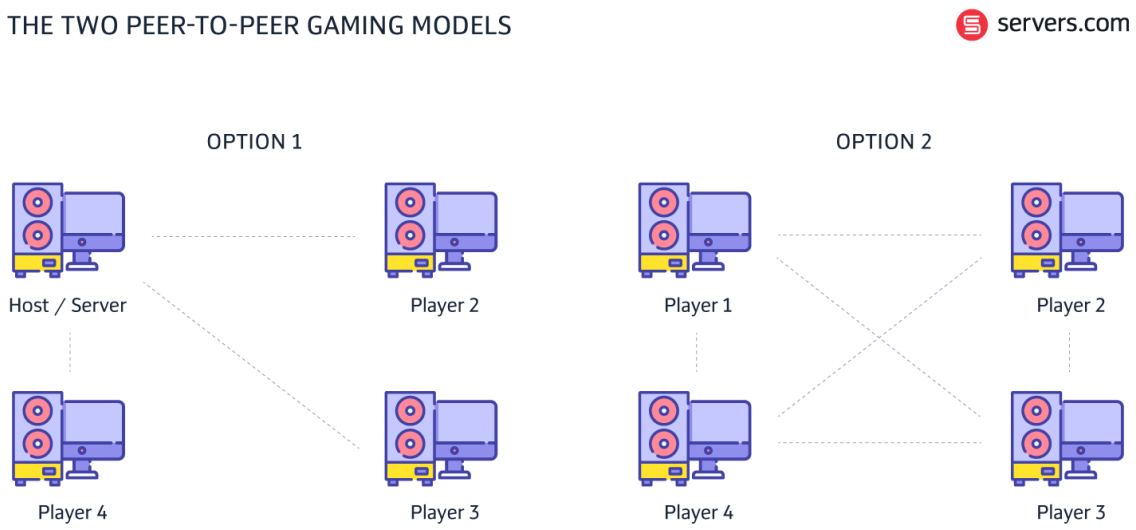
moninpelit alkoivat yleistymään arcade-pelihalleissa ympäri maailman. Ensimmäiset useammalla päätelaitteella pelattavat saman verkon moninpelit valmistettiin heti seuraavina vuosina opetus tietokonejärjestelmissä, mutta tavallisilla kotikonsoli kuluttajilla ei ollut pääsyä niihin vielä vuosiin. (Dreher 2020, 411-418.)

2.1 Paikallisista moninpeleistä verkkoon

Ensimmäiset useamman päätelaitteen moninpelit kuluttajien saataville valmistettiin verkkoyhteyksien hitaan nopeuden ja kalliin hinnan takia lähiverkossa. Amerikkalainen Kesmai-yritys julkaisi Island of Kesmai-verkkomoninpelin jo vuonna 1985, mutta se ei se kerännyt isoa pelaajakuntaa hintansa takia, sillä sen pelaaminen maksoi halvemman nettikaistan käyttäjälle 6 dollaria tunnilta (Mulligan & Patrovsky 2003). Ensimmäinen paikallisen verkon moninpeli kuluttajien saataville ilmestyi vuonna 1986, kun Microsoft julkaisi Flight simulator 2:sen Atari ST- ja Commodore Amiga-järjestelmille, joka salli kahden pelaajan yhdistämisen paikallisesti samaan peliin toistensa kotikonsoleihin modeemilla tai serial-kaapelilla (Amiga Flight Simulator II 1988). Useamman kuin kahden päätelaitteen moninpeli ilmestyi kuluttajille lähiverkon kehittyessä vuonna 1991, kun Apple Macintosh julkaisi Spectre-pelin Macintosh-järjestelmälle (Spectre 1991 Video Game n.d). Kotikoneiden verkkomoninpelit saivat paljon isomman suosion vasta myöhempinä vuosina, kun verkon käyttämisen hinta oli paljon kuluttaja ystävällisempi. Huomattavimmat verkkomoninpelien pelaajamäärä kasvut alkoivat MMO-pelien grafiikkojen kehittyessä vuonna 1997, kuten amerikkalaisen Origin Systemsin julkaisema Ultima Online MMO-peli keräsi 100 000 maksavaa asiakasta puolessa vuodessa (Ultima's Population Reaches 100,000 2012). Verkkomoninpelit kokivat räjähtävän kasvun 2000-luvun vaihteessa, kun moninpeli kehittäjät julkaisivat monta todella hyvin vastaanotettua verkkomoninpeliä, kuten Counter Strike (1999), Runescape (2001), Battlefield 1942 (2002) ja World Of Warcraft (2004).

Kaikki ensimmäiset useamman päätelaitteen moninpelit kuluttajien saataville toimivat vertaisverkkoina, jossa kaikki pelaajat pyörittivät omaa simulaatiota pelistä ja lähettivät sitä toisillensa (What is a gaming server? 2023). Ensimmäinen menestyvä asiakas-palvelin-mallinen peli ilmestyi vuonna 1986, kun avoimen lähdekoodin Xtrek-peli julkaistiin X Windows-järjestelmälle (McFadden 1994). Asiakas-palvelin-mallisessa moninpelissä kaikki päätelaitteet vastaanottavat yhdistetyn palvelimen simulaatiota, joka muuttuu päätelaitteiden lähettämien syötteiden mukaisesti (Client-Server 2023).

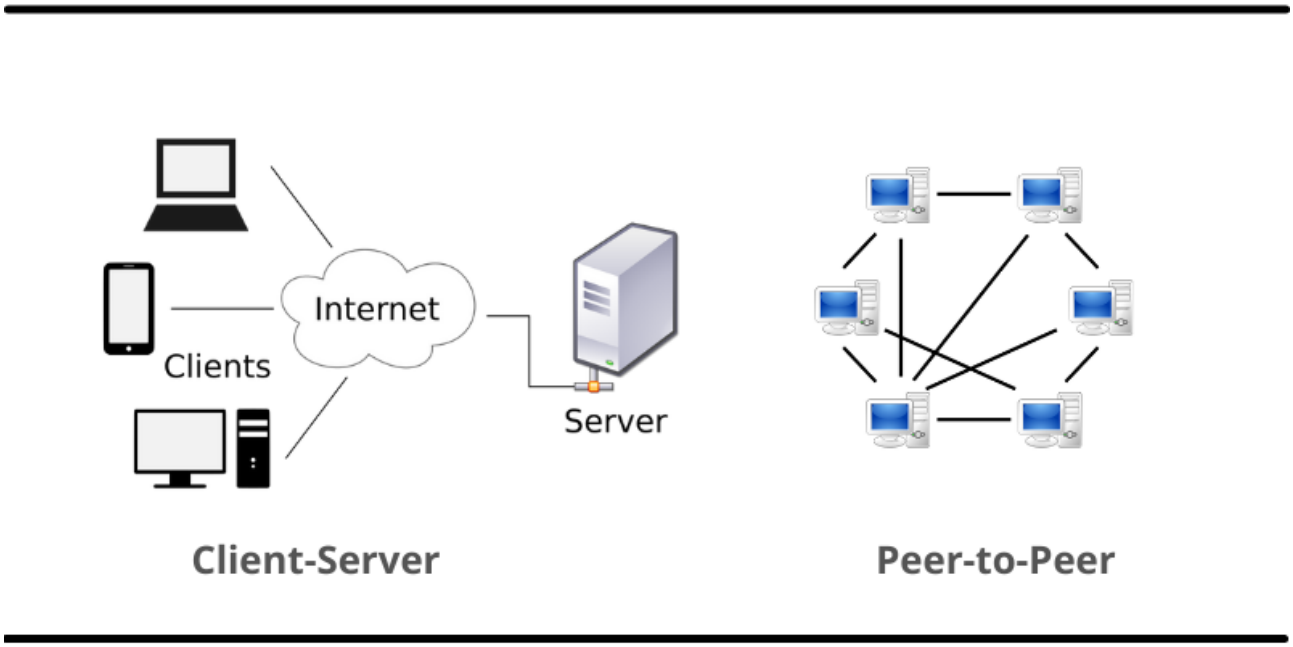
Vertaisverkko tarkoittaa verkon ohjelmistoarkkitehtuuria, jossa kaikki verkkoon yhdistäneet tahot toimivat vertaisina keskenään samoilla oikeuksilla, eli jokainen verkkoon yhdistänyt laite toimii samaan aikaan asiakkaana ja palvelimena muille verkon jäsenille. Vertaisverkossa jokainen yhdistänyt vertainen jakaa osan kaistasta, prosessointitehosta ja muistista keskenään (Kanade 2023). Pelinkehityksessä vertaisverkolla tarkoitetaan moninpelin verkko rakennetta, jossa ei ole pelaajien ulkopuolista palvelinta, vaan peliin yhdistäneet pelaajat toimivat itse isännöitsijöinä. Moninpelien vertaisverkossa on kaksi mahdollista verkko rakennetta, joista toinen toimii vertaisverkko ohjelmistoarkkitehtuurin mukaisesti jakaen isännyyden, yhteyden ja suoritustehon yhdistäneiden pelaajien välillä (Differences between peer to peer and dedicated game server hosting 2022). Toinen moninpelien vertaisverkon verkko rakenne tekee yhdestä yhdistäneestä pelaajasta isännän, joka välittää yhdistäneiden pelaajien tapahtumat toisilleen palvelinmaisesti (Kuva 1).



Kuva 1. Vertaisverkon kaksi verkko rakennetta moninpeleissä

Asiakas-palvelin-malli tarkoittaa verkon ohjelmistoarkkitehtuuria, jossa kaikki verkkoon yhdistäneet tahot toimivat määritellyn palvelimen asiakkaina (Kuva 2). Verkon palvelimella on aina asiakkaista eroava palvelinohjelmisto, joka tarjoaa asiakkaille tiettyä palvelua tai toimintoa. Palvelin jakaa asiakas-palvelin-mallisessa verkossa resurssinsa yhdistäneiden asiakkaiden kanssa, jolloin asiakkaiden ei tarvitse kuin kommunikoida palvelimen kanssa kuormittumatta itse (Client-Server Model 2022). Pelinkehityksessä asiakas-palvelin-mallilla tarkoitetaan moninpelin verkon

rakennetta, jossa kaikki pelaajat yhdistävät pelipalvelinohjelmiston peli-instanssiin. Pelipalvelinohjelmisto pyörittää isäntä peli-instanssia, johon pelipalvelin vastaanottaa yhdistäneiden pelaajien syötteitä ja lähettää instanssia yhdistäneille pelaajille (Tsaruk 2023). Pelipalvelinohjelmistoa suoritetaan tavallisesti erillisellä pelipalvelimella riittävien resurssien takaamiseksi yhdistäneille pelaajille, mutta sitä voidaan pyörittää myös reititetyltä kotikoneelta. Pelipalvelinohjelmisto on aina erillinen itse pelistä, joten pelipalvelinohjelmistoa pyörittävän laitteen pitää suorittaa erillistä peli-instanssia liittyäksensä samalla laitteella pyörivään pelipalvelinohjelmiston peli-instanssiin.



Kuva 2. Asiakas-palvelin-mallinen verkko ja vertaisverkko

2.2 Tila ja syöte pelipalvelimella

Pelipalvelimenohjelmisto asiakas-palvelin-mallisessa moninpelissä pyörittää yhtä ja oikeata peli-instanssia, joka vastaanottaa yhdistäneiden päätelaitteiden yhteydet ja lähettää tämän ainoan oikean peli-instanssin samanlaisena kaikille päätelaitteille. Päätelaitteet lataavat omasta muistista staattiset ja dynaamiset pelin osat omaan instanssiinsa, joista muodostuu päätelaitteen puoleinen uniikki peli-instanssi. Päätelaitteiden uniikki peli-instanssi vastaanottaa palvelimelta oikean peli-instanssin dataa, josta peli yrittää muodostaa jokaiselle päätelaitteelle oikean näköisen peli-instanssin asettamalla dynaamiset osat oikean peli-instanssin mukaisille paikoille. (Client-Server 2023.)

Käytännössä päätelaitteiden peli-instanssit kommunikoivat palvelimen peli-instanssin kanssa jokaisen palvelimen pelisimulaation tickraten mukaisen tickin aikana. Jos yhden tickin aikana ei tule syötettä päätelaitteelta, päätelaite vain vastaanottaa peli-instanssin dataa palvelimelta, mutta jos päätelaite antaa syötettä tickin aikana, päätelaite myös lähettää oman peli-instanssinsa syötettä palvelimen peli-instanssin pelaajalle määriteltyyn ohjaimeen. Palvelimen peli-instanssia voi siis ajatella kaikkien päätelaitteiden oikeana peli-instanssina, jota jokainen päätelaite ohjaa lähettämällä syötettä omasta peli-instanssistansa palvelimella sijaitsevaan ohjain osaansa pelin tickien aikana. (Source Multiplayer Networking n.d.)

2.3 Epädeterministinen ja deterministinen pelimoottori

Pelimoottori on ohjelmistokehys, joka on suunniteltu tukemaan sen päälle ohjelmoitavaa peliä. Pelimoottori koostuu usein renderöinti- ja fysiikkamoottorista, ääni, animaatio, video, AI, verkotus, muistinhallinta, threading ja lokalisaatio tuesta. Pelimoottorille rakennetaan kohtauksia ja tiloja, joista pelaaja etenee uusiin kohtauksiin pelaamalla tilan tietynlaiseksi pelin sisäisillä dynaamisilla komponenteilla ja ohjelmoiduilla mekaniikoilla.

Epädeterministinen pelimoottori tarkoittaa pelimoottoria, joka voi kohtauksen sisällä edetä erilaiseen tilaan samoilla syötteillä. Erilaiseen tilaan päätyminen johtuu yleensä rajattomien liukulukujen erilaisiin pyöristyksiin päätymisestä tietokoneen rajallisten bittijonojen edetessä eri kohdissa tietokoneen ohjelmistoa. Deterministinen pelimoottori tarkoittaa pelimoottoria, joka päätyy kohtauksen sisällä aina samaan tilaan saman syötteen saadessa.

3 Moninpelien ongelmia ja ratkaisuja

Moninpelien kehityksessä on ollut haasteita ensimmäisten moninpelien kehityksestä asti, joita on yritetty korjata ja parantaa erilaisilla verkko arkkitehtuureilla ja ohjelmallisilla tekniikoilla. Moninpelien kehityksen suurimpia haasteita ovat tavallisesti netcode-ohjelmoinnin tuoma monimutkaisuus tavallisen peliohjelmoinnin päälle, pelipalvelun skaalautuvat isännöintiongelmat, yhdistäneiden pelaajien yhteyksien eri nopeudet ja pelaajien välinen synkronointi. (Multiplayer networking challenges 2023.)

Netcode-ohjelmoimisen suurin ero tavalliseen ohjelmointiin on netcode-koodin suorituslogiikka, sillä sen sijaan että yksittäinen pelaaja suorittaa kaiken koodin tapahtumissa itse, jokainen pelaaja suorittaa omia tapahtumia, jotka pitää lähettää toisille pelaajille suoritettavaksi heidän puolellensa. Moninpelissä on aina suunniteltu isännöintimuoto, joka aiheuttaa eritasoisia etuja ja ongelmia eri osa-alueissa isännöintimuodon mukaan (Andrea n.d.). Vertaisverkko isännöintimuotona poistaa tarpeen erillisille palvelimille, mutta on riippuvainen isäntänä toimivista pelaajista ja on hankalasti skaalautuva suuremmalle pelaajamäärälle yhteen peliin. Asiakas-palvelin-malli isännöintimuotona on paljon paremmin skaalattavissa suuremmille pelaaja määrille yhteen peliin, mutta on riippuvainen palvelimen sijainnista pelaajiin nähden, voi estää pelaamisen palvelinongelmien aikana ja on hintava erillisten palvelimien vuokrauksen tai ostamisen takia.

3.1 Vertaisverkon etuja ja ongelmia moninpeleissä

Vertaisverkon suurimpia etuja pelipalvelun isännöintimuotona pelinkehittäjille on isännöintimuodon hinta, sillä pelille ei tarvitse vuokrata tai ostaa erillisiä palvelimia pelaajan toimiessa isäntänä muille pelaajille. Pelinkehittäjien suurena etuna vertaisverkossa on myös pelin päivityksien helppous, sillä pelinkehittäjien ei tarvitse erikseen sulkea ja päivitellä pelipalvelimien ohjelmistoja, vaan peliä voi päivittää suoraan pelaajille.

Vertaisverkon suurimpia etuja pelaajille ovat pienet viiveet ja vasteajat, pelaajien yhteyksien ollessa nopeita. Vertaisverkko mahdollistaa minimaalisen viiveen pelaajien välille suoran yhteyden ansiosta, sillä pelaajien ei tarvitse lähettää tietoa ja odottaa vastausta erillisen palvelimen kautta. Jaetun isännyyden vertaisverkossa moninpeli skaalautuu hyvin ja tehokkaasti jakaen työkuormaa kaikkien vertaisten välillä pelaajamäärän pysyessä kohtuullisena, mutta pelaajamäärän kasvaessa liian suureksi ilman kattavaa pelin sisäistä tiedonhallintaohjelmointia kasvaa kaikkien viiveet ja vasteajat suuren tiedonmäärän liikkumisen takia pelaajien välillä (Neumann 2007).

Vertaisverkon suurimpia pelaajapuolen ongelmia ovat pelaajien yhteyksien vaihtelevuus, skaalautuvuus ja turvallisuusongelmat. Yhden isännän vertaisverkko moninpelissä isäntäyhteys määrittää kaikkien yhdistäneiden pelaajien yhteyksien nopeudet, eli jos isäntäyhteydellä on suuri viive, on sama viive kaikilla vertaisverkossa kommunikoivilla yhteyksillä. Jaetun isännyyden vertaisverkko moninpelissä ei yksittäisen pelaajan yhteys vaikuta pelikokemuksen vakauteen yhtä paljon, mutta usean pelaajan yhteysongelmat vaikuttavat kaikkien pelaajien yhteyksiin muuttaen pelin

epävakaaksi ja nopeuden vaihtelevaksi. Vertaisverkoissa on aina pitänyt ottaa turvallisuus erikseen huomioon, sillä pelaajien välissä ei ole tiedostoihin pääsyn oikeuksista huolehtivaa palvelinta. Vertaisverkko aiheuttaa myös ongelmia pelinkehittäjille, sillä kaikki pelien sisäiset tapahtumat ja kaatumiset pitää erikseen lähettää pelaajien koneilta kehittäjille palvelimien puutteen takia. (Neumann 2007.)

3.2 Asiakas-palvelin-mallisen verkon etuja ja ongelmia moninpeleissä

Asiakas-palvelin-malli ratkaisi monta vertaisverkon ongelmaa, kuten suuren pelaajamäärän synkronoinnin, pelaajien eritasoisten yhteyksien toisiin vaikuttamisen ja turvallisuuden. Asiakas-palvelin-malli mahdollisti pelaamisen yleistymisen verkon välityksellä, sillä lähetettävän ja vastaanotettavan datan määrä pieneni huomattavasti, sillä jokaisen pelaajan piti kommunikoida palvelimen kanssa vain muutokset. Asiakas-palvelin-mallisen moninpelin suurin ongelma oli yhdistäneiden päätelaitteiden eri viiveet palvelimen kanssa, mutta pian asiakas-palvelin-mallisten moninpelien yleistyessä keksittiin erilaisia ohjelmallisia ratkaisuja ongelmiin, kuten asiakapuolen ennustus-, viiveen korvaus- ja sovittelutekniikoita, jotka paransivat moninpeli kokemusta entisestään (Bernier 2001).

Asiakas-palvelin-mallisen verkon suurimpia etuja pelipalvelun isännöintimuotona pelinkehittäjille on helpompi konfigurointi ja ongelmanratkaisu, sillä keskitetyn palvelimen kanssa on helpompi kommunikoida kuin erillisten isäntänä toimivien pelaajien kanssa. Pelinkehittäjien etuna on myös helpommin toteutettava skaalautuvuus moninpelille, sillä palvelimena toimivan laitteen tehot on helpompi mitoittaa pelin vaatimiin tehoihin ja pelaajamääriin. Turvallisuus on myös suuri etu asiakas-palvelin-mallisessa moninpelissä, sillä kaikki tiedot ovat varastoitu palvelimelle ja pelaajat ovat yhteyksissä toisiinsa vain palvelimen välityksellä.

Asiakas-palvelin-mallisen moninpelin suurin etu pelaajille turvallisuuden lisäksi on tasapainoinen ja toisista pelaajista riippumaton pelikokemus. Palvelin käsittelee kaikkien pelaajien yhteyksiä tasapuolisina ja erikseen, eli jokaisella verkon yli yhdistäneellä pelaajalla on samantasoinen palvelinpuolen viive ja yksittäisiin pelaajiin ei vaikuta toisten pelaajien vaihtelevat yhteydet. Asiakas-palvelin-mallinen moninpeli on pelaajille myös suoritustehon puolesta tasapainoinen, sillä pelaajien omat suoritusteho ongelmat eivät vaikuta muiden pelaajien pelin toimivuuteen.

Asiakas-palvelin-mallisen moninpelin suurimpia ongelmia pelaajille ovat palvelimien sijainti ja saatavuus, sillä palvelimet voivat olla liian kaukana pelaajaan nähden aiheuttaen ison viiveen tai palvelimet voivat olla alhaalla virheen, kaatumisen tai päivityksen seurauksena. Palvelin voi olla myös saavuttamattomissa liian monen yhteyden yhtäaikaisen vastaanottamisen tai verkkohyökkäyksen seurauksena. Pelinkehittäjien suurimmat ongelmat asiakas-palvelin-mallisessa moninpelissä tulevat palvelimien tyyriistä hinnoista, sillä tarvittavat palvelimet pitää ostaa tai vuokrata, sekä huoltaa ja ylläpitää. Palvelimilla on usein oma työntekijä vahtimassa, ettei mikään palvelin ole rikki tai alhaalla kaatumisen tuloksena. (Client-Server Network: Definition, Advantages, and Disadvantages 2023.)

3.3 Ohjelmalliset korjaukset moninpelien ongelmiin

Verkkomoninpelien ongelmia ja haasteita on pyritty korjaamaan erilaisilla verkko-ohjelmointimenetelmillä, kuten asiakapuolen ennustus-, viiveen korvaus- ja sovittelutekniikoilla. Erilaiset moninpelikokemusta parantavat verkko-ohjelmointi menetelmät alkoivat kehittymään vasta asiakas-palvelin-mallisten verkkomoninpelien yleistyessä, sillä vertaisverkon viiveongelmat pystyttiin minimoimaan nopeammalla yhteydellä suoraan toisiinsa kytköksissä olevien pelaajien ansiosta. Asiakas-palvelin-mallisessa verkkomoninpelissä kaikkien pelaajien syötteet ja tapahtumat pitää lähettää palvelimelle, palvelimen pitää prosessoida tapahtumat luoden uuden pelitilan ja lähettää tämä uusi pelitila pelaajille luoden paljon isomman viiveen kuin vertaisverkossa.

Asiakaspuolen ennustus tarkoittaa verkko-ohjelmointimenetelmää, jossa asiakkaan puolella suoritetaan syötteistä päättelevää koodia. Asiakaspuolen ennustus takaa pelaajalle sulavamman kokemuksen, sillä pelaajalle peli reagoi nopeammin, kun ei tarvitse odottaa palvelimelta syötteen mukaista uutta pelitilaa ennen etenemistä. Asiakaspuolen ennustuksessa pelaajan syöte käsitellään palvelimen lisäksi pelaaja puolella, jonka seurauksena pelaajan pelitila päivittyy paikallisesti ennustaen palvelimelta tulevaa syötteen mukaista uutta pelitilaa. Palvelimelta tullut uusi päivittynyt pelitila on yleensä identtinen pelaajan syötteistä paikallisesti ennustetun pelitilan kanssa, mutta voi olla erilainen riippuen pelin epädeterministisyydestä ja viiveen määrästä. Paikallisesta eroavan pelitilan saapuessa palvelimelta peli voi joutua desynkronoituun tilaan, jossa paikallinen pelitila ei vastaa palvelimen pelitilaa tehden pelistä pelaamiskelvottoman. (Prediction 2023.)

Palvelimen sovittelu on tekniikka, jolla vältetään asiakaspuolen ennustuksesta aiheutuvaa desynkronoituun tilaan joutumista. Palvelimen sovittelussa jokaiseen pelaajalta lähtevään syötteeseen lisätään suoritusjärjestystunniste palvelimen luettavaksi, jonka avulla kaikki yhteyden nopeudesta riippuvat pelitilojen eroavaisuudet minimoidaan. Ilman palvelimen sovittelun suoritusjärjestystunnistetta, kaksi peräkkäistä syötettä pienelläkin yhteyden viiveellä voisi saada paikallisen ja palvelimen pelitilan eroamaan toisistaan. (Reconciliation in multiplayer games 2023.)

Viiveen korvaus on tekniikka, minkä avulla palvelin katsoo pelaajan toimintoja pelaajalle sen hetkisen näytetyn maailman perusteella. Ilman viiveen korvausta isomman viiveen pelaaja ei voisi vuorovaikuttaa palvelimen pelimaailman kanssa halutusti, sillä palvelimen pelimaailma olisi edennyt erilaiseen tilaan pelaajan tapahtumien rekisteröityessä palvelimelle. Viiveen korvaus mahdollistaa responsiivisen palvelinpohjaisen pelin tunteen, sillä pelaaja tuntee vuorovaikuttavansa suoraan näkemänsä pelitilan kanssa, eikä vain palvelimella sijaitsevan pelitilan. (Lag Compensation n.d.)

4 Unity ja Exit Games Photon

Unity on C-kieliin pohjautuva amerikkalaisen Unity Technologiesin vuonna 2005 kehittämä monialusta pelimoottori, joka on nykypäivänä yksi eniten käytössä oleva pelimoottori pelinkehityksessä (Toftedahl 2019). Unity-pelimoottorin alkuperäinen idea oli tehdä pelinkehityksestä helpommin lähestyttävää kaikille (Axon 2016). Unity-pelimoottori julkaistiin alun perin vain Mac OS X-järjestelmälle, mutta tuki web-selaimelle, Microsoft Windows-, IOS- ja Android-järjestelmille lisättiin seuraavina vuosina. Unity-pelimoottori on ollut kaikista käytetyin mobiilipelimoottori maailmassa vuodesta 2012 lähtien (Game Developer May 2012 Issue 2012).

Photon Engine on saksalaisen Exit Gamesin vuonna 2003 kehittämä monialusta moninpeliverkkomoottori kehitystyökalujen ja palveluiden tarjoaja. Photonin kehitystyökalut ovat reaaliaikaisia moninpeli ohjelmistokehyksiä, jotka koostuvat palvelimesta ja useammasta asiakasohjelmiston kehityspaketista suurille alustoille. Photonin palvelut koostuvat kommunikointipalvelinohjelmistoista ja verkkomoottori tuen tilauksista, joiden avulla saa pääsyn Photonin keskustelupalstoille, koodinäytteisiin ja suoraan apuun Photonin työntekijöiltä. (Photon n.d.)

4.1 Unity-pelimoottori moninpeleissä

Unity-pelimoottori on tarkoitettu joka tarkoituksen ja alustan pelimoottoriksi, mutta se tukee suoraan asennuksen jälkeen vain yksinkertaisia yksinpelejä. Unity-pelimoottoria on tarkoitus käyttää asentamalla jokaiseen työstettävään projektiin erilliset Unityn tai kolmannen osapuolen tarjoamat paketit tai kirjastot, jotka mahdollistavat juuri sen projektin ominaisuudet ja toiminnot. Pakettien ja kirjastojen erikseen asentaminen tarpeen mukaan mahdollistaa Unity-pelimoottorin pienen koon projektissa ja varmistaa kevyen pelimoottorin nopeuden. (Tuliper 2015.)

Unity tarjoaa omana kirjastopakettina kahdenlaista moninpeli ratkaisua kehittäjien käyttöön, valmista pelipalvelinisännöintipakettia nimeltä Multiplay ja itse ohjelmoitavia netcode-kirjastopaketteja tarpeen mukaan, Netcode for Gameobjects (NGO) ja Netcode for Entities (NGE). Unity tarjoaa omina kirjastopaketteina myös alhaisemman tason verkkokirjastoa Unity Transport ja moninpelien kehitystyökalua Unity Multiplayer Tools, joka sisältää profiloijan, suoritusaika tilastomonitorin, verkkosimulaattorin ja verkkokohtaus visualisoijan.

4.1.1 Unity Multiplay

Unity Multiplay on Unityn tarjoama kirjastopaketti minkä vain pelimoottorin päälle, jonka projektiin asentamisen jälkeen kehittäjä saa käyttöönsä visuaalisen käyttöliittymän hallinnoimaan moninpelin toiminnallisuuksia. Unity Multiplay on valmis paketti ohjaamaan kaikkia moninpeli toiminnallisuuksia, kuten palvelin määrän skaalautuvuutta, matchmaking-palveluita, moninpelianalytiikkoja ja pelaajien sijoittelua palvelimille sijainnin mukaan. (Game server hosting (Multiplay) n.d.)

Unity Technologies otti Unity Multiplayn käyttöön ostettuaan Multiplayn Game Digitalilta vuonna 2017, josta kaikki Multiplayn parissa työskennelleet työntekijät siirtyivät Unitylle töihin (Unity Technologies Acquires Game Hosting Division of Multiplay from GAME Digital, PLC 2017). Unity Multiplayn tarkoitus on tarjota yksinkertaista moninpelikehitys API:a, jossa kehittäjien ei tarvitse itse kirjoittaa netcodea.

4.1.2 Unity netcode-paketit

Unity netcode-paketit ovat Unityn tarjoamia kirjastopaketteja Unity-pelimoottoria käyttäviin projekteihin, joiden avulla kehittäjät voivat itse ohjelmoida tarvitsemansa netcode-toiminnallisuudet projektiin. Unity tarjoaa kahta erilaista netcode-kirjastopakettia asennettavaksi, Netcode for GameObjects- ja Netcode for Entities-pakettia. Molemmilla netcode-paketeilla voi toteuttaa samanlaisia pelejä, mutta paketit on tarkoitettu erilaisiin peleihin.

Netcode for GameObjects-paketti on tarkoitettu pienen mittakaavan hidastempoisille ja rennoille peleille, missä verkotettujen kappaleiden ei tarvitse olla äärimmäisen tarkkoja ja nopeita, eikä pelissä ole välttämätöntä tarvetta pelipalvelimille tai huijauksen estoon. Netcode for Entities-paketti on tarkoitettu suuremman mittakaavan kilpailullisille peleille, missä on tarvetta tarkoille ja nopeille vasteajoille pelaajien välille. Netcode for Entities-paketti perustuu entiteettikomponenttijärjestelmään ja tarjoaa useita ohjelmallisia korjauksia verkkomoninpelien ongelmiin. (Build multiplayer games with Unity netcode n.d.)

4.2 Exit Games Photon Enginen tuotteet

Photon Enginen kaikki tuotteet ovat tarkoitettu reaaliaikaisten moninpelien kehitys tarkoitukseen ja toimivat alustojen ristiin pelaamisessa (Realtime Intro n.d). Photon tarjoaa pääsääntöisesti kolmea eri tuotetta, Fusion, Quantum ja Realtime. Photon tarjosi aikaisemmin suuressa suosiossa olleita Pun ja Bolt nimisiä palveluita, mutta näistä on lopetettu tuki ja ne ovat jääneet legacy-tuotteiksi, sillä niiden arkkitehtuuri ei sallinut nykyaikaisempaa optimointia (Pun & Bolt vs Fusion n.d.).

4.2.1 Photon Fusion

Photon Fusion on tilan siirto ja synkronointi verkkokirjastopaketti Unityyn. Fusion suunniteltiin nykyaikaistamaan ja yhdistämään Pun- ja Bolt-ohjelmistopaketit, sillä vanhemmilla paketeilla oli ongelmia nykyaikaisten moninpelitoimintojen kanssa, kuten suurien pelaajamäärien, pelaajien toimintojen tarkkaan jäljentämisessä ja oikeuksien käsittelyissä (Pun & Bolt vs Fusion n.d.).

Photon Fusion on suunniteltu toimimaan erittäin helppokäyttöiseksi Unity-editorissa, sillä suuri osa toiminnallisuudesta toimii lisäämällä Unity-kappaleisiin Fusion-skriptitiedostoja komponenteiksi, joita pystyy tiedoston koodin muuttamisen lisäksi muokkaamaan editorissa komponentin

julkisia arvoja muuttamalla. Fusion tarjoaa Unity-tyylisen työnkulun lisäksi datan kompressointia, asiakaspuolen ennustus- ja viiveen korvaustekniikoita valmiiden komponenttien muodossa. Fusionin datan kompressointi tapahtuu kaistanleveyden käyttöä vähentävällä algoritmilla, joka siirtää tilojen tietoja pakattuina tilannekuvina tai osittaisina paloina lopullisella yhtenäisyydellä. (Fusion 2 Introduction n.d.)

4.2.2 Photon Quantum

Photon Quantum on nopean suorituskyvyn täysin deterministinen entiteettikomponenttijärjestelmäkehys Unity-verkkomoninpeleille. Quantum perustuu ennustus/peruutus lähestymistapaan, mikä on ihanteellisen latenssiherkille verkkomoninpeleille. Quantum toimii käytössä hyväksi todettujen Photonin tuotteiden ja infrastruktuurin päällä, kuten Photon Realtime kuljetuskerroksen ja Photon-palvelimien liitännäisten päällä isännöimään palvelinlogiikkaa.

Tavallisissa deterministisissä järjestelmissä pelin päätelaitteet odottavat jokaisen pelaajan syötettä ennen etenemistä. Quantum-pelissä päätelaitteet etenevät vapaasti paikallisessa simulaatiossa käyttäen myös saapuvien pelaaja syötteiden kanssa ennustus/peruutus järjestelmää, joka palauttaa ja uudelleen simuloi pelin tilaa väärien ennusteiden tapahtuessa. (Quantum 2 Intro n.d.)

4.2.3 Photon Realtime

Photon Realtime on pohjakerros verkkomoninpeleille ja korkeamman tason verkkoratkaisuille melkein mille vain alustalle. Realtime ratkaisee skaalautuvalla lähestymistavalla verkkomoninpelien ongelmia, kuten pelaajien välistä kommunikointi- ja matchmaking-nopeutta. Kaikki Photonin tuotteet käyttävät Realtime-pohjakerrosta alemmalla tasolla. Realtime sisältää API-kehyksiä, ohjelmatyökaluja ja palveluja, jotka ohjaavat käyttäjien ja palvelimien välisiä vuorovaikutuksia.

Photon Realtime on asiakaspuolen ohjelma, mitä pelit ja ohjelmat pystyvät käyttämään suoraan, ja missä käyttäjät voivat vuorovaikuttaa keskenään melkein millä vain alustalla. Realtime toimii yhdistämällä kaikki päätelaitteet omistettujen palvelimien sarjaan, jotka koostuvat kolmen eri tehtävän palvelimista, autentikointi- ja alueellisen jakelupalvelimiin, matchmaking-palvelimiin ja pelipalvelimiin. (Realtime Intro n.d.)

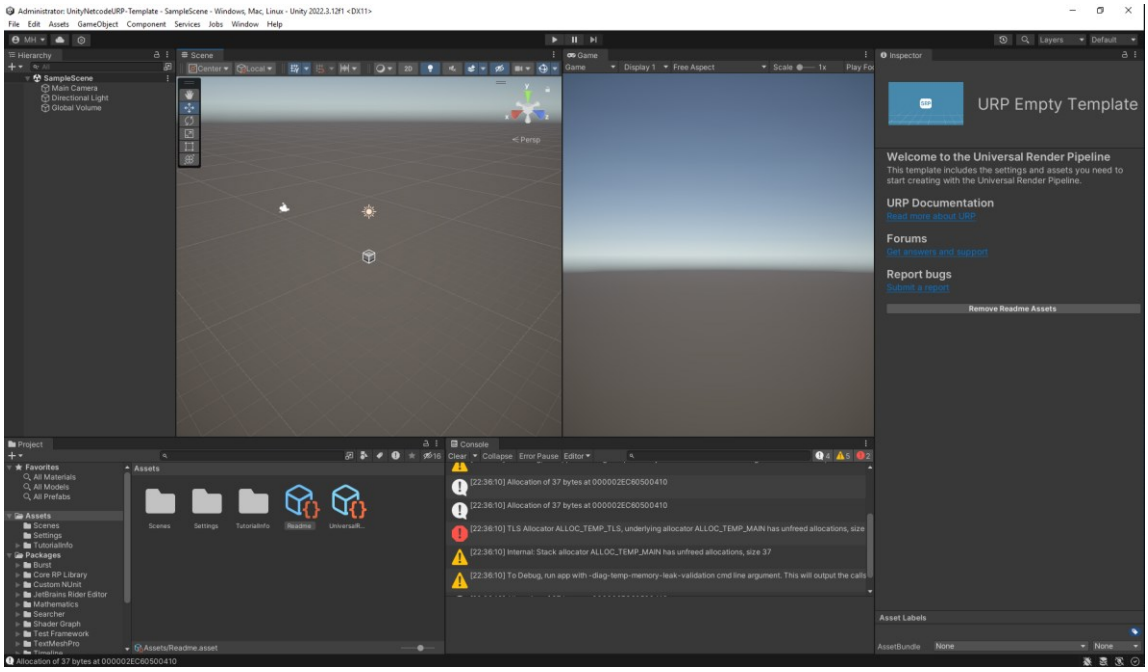
5 Netcode for Entities prototyypin toteutus

Tässä opinnäytetyössä tehtiin kaksi samanlaista verkkomoninpeliä Unity-pelimoottorille kahdella eri entiteettikomponenttijärjestelmäpaketilla, toinen toteutettiin Unityn omalla Unity Netcode for Entities-paketilla ja toinen Photon Enginen täysin deterministisellä Quantum-paketilla. Prototyyppien tarkoituksena oli saada mahdollisimman samanlaisista tilanteista vertailu dataa, josta näkisi mahdollisimman selkeästi täysin deterministisen pelimoottorin tuomat tehokkuudet moninpeli toteutuksessa.

Unity Netcode for Entities-verkkomoninpeliprototyypin toteutus tehtiin tyhjän Unity-projektin päälle Unityn omien Netcode For Entities-dokumentaatioiden mukaisesti. Prototyypin tarvittavan toiminnallisuuden tekemistä jatkettiin itsenäisesti dokumentaation mukaisen pohjan valmistuttua (Getting started n.d.).

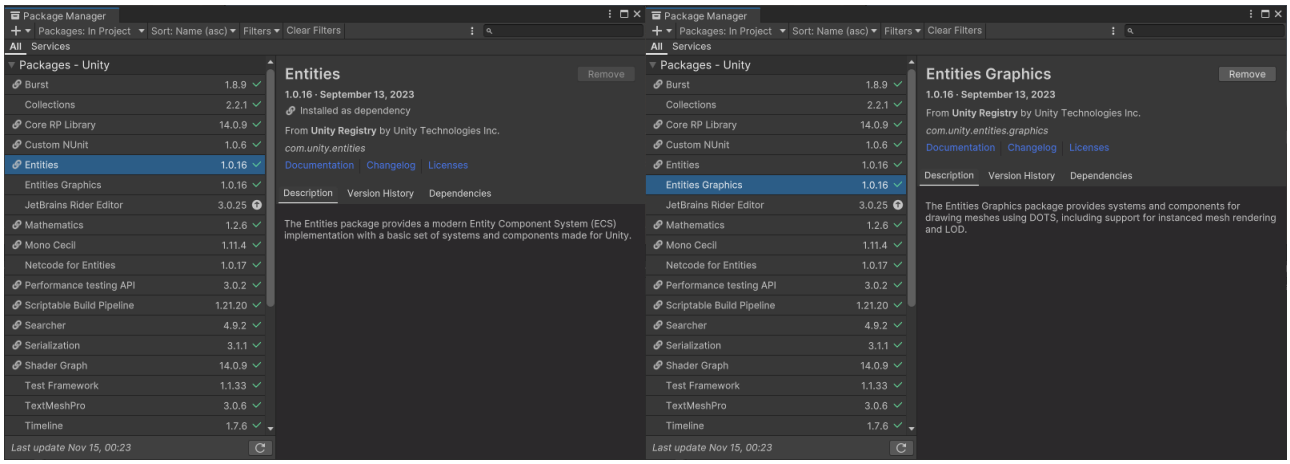
5.1 Entities-projektin pohja

Unity Netcode for Entities-prototyyppiprojekti aloitettiin tekemällä uusi, tyhjä Unity-projekti Unityn omasta URP-Templaatista (kuva 3).



Kuva 3. Tyhjä Unity-projekti luotuna Unityn omasta URP-templaatista

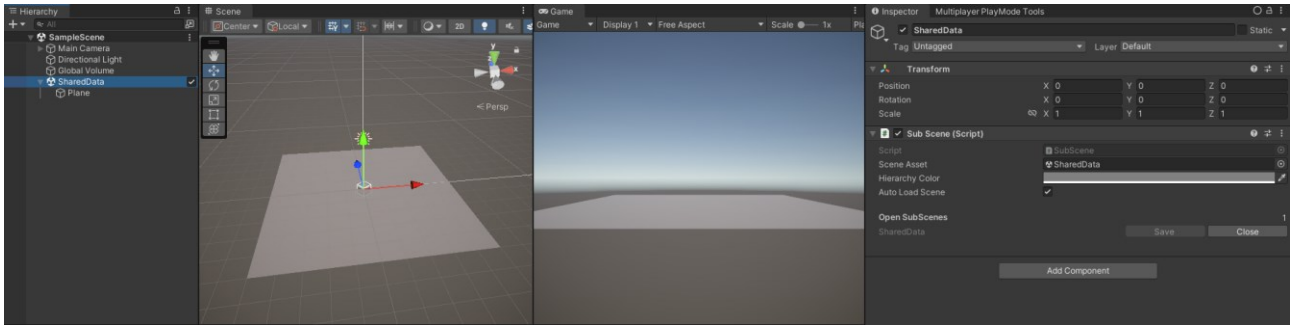
Tyhjään Unity-projektiin lisättiin tarvittavat paketit Unityn package managerilla. Lisättävät paketit sisälsivät kaiken Netcode For Entities-verkkomoninpelin toiminnallisuudesta ja käyttöönotosta. Paketit lisättiin nimillä com.unity.netcode ja com.unity.entities.graphics. (kuva 4.)



Kuva 4. Tarvittavat paketit Unity package managerissa

5.2 Entities-projektin maailman alustus

Tyhjään projektiin lisättiin SharedData-niminen alakohtaus, jonka sisällä tapahtui kaikki pelaajille verkon yli synkronoitava sisältö. SharedData-nimisen alakohtauksen lapseksi lisättiin 3D-kappale kentän maatasoksi (kuva 5).



Kuva 5. Projektin kohtaukseen lisätty alakohtaus ja maa tasona toimiva 3D-kappale

Projektiin lisättiin Game.cs-tiedosto, joka suorittaa sovelluksen käynnistyessä Entities-paketin ClientServerBootstrap.cs-tiedoston Initialize-metodin asetetulla porttinumerolla (Kuva 6).

```
using System;
using Unity.Entities;
using Unity.NetCode;
using Unity.Networking.Transport;

[UnityEngine.Scripting.Preserve]
0 references
public class GameBootstrap : ClientServerBootstrap
{
    0 references
    public override bool Initialize(string defaultWorldName)
    {
        AutoConnectPort = 7979;
        return base.Initialize(defaultWorldName);
    }
}
```

Kuva 6. Lisätty Game.cs-tiedosto

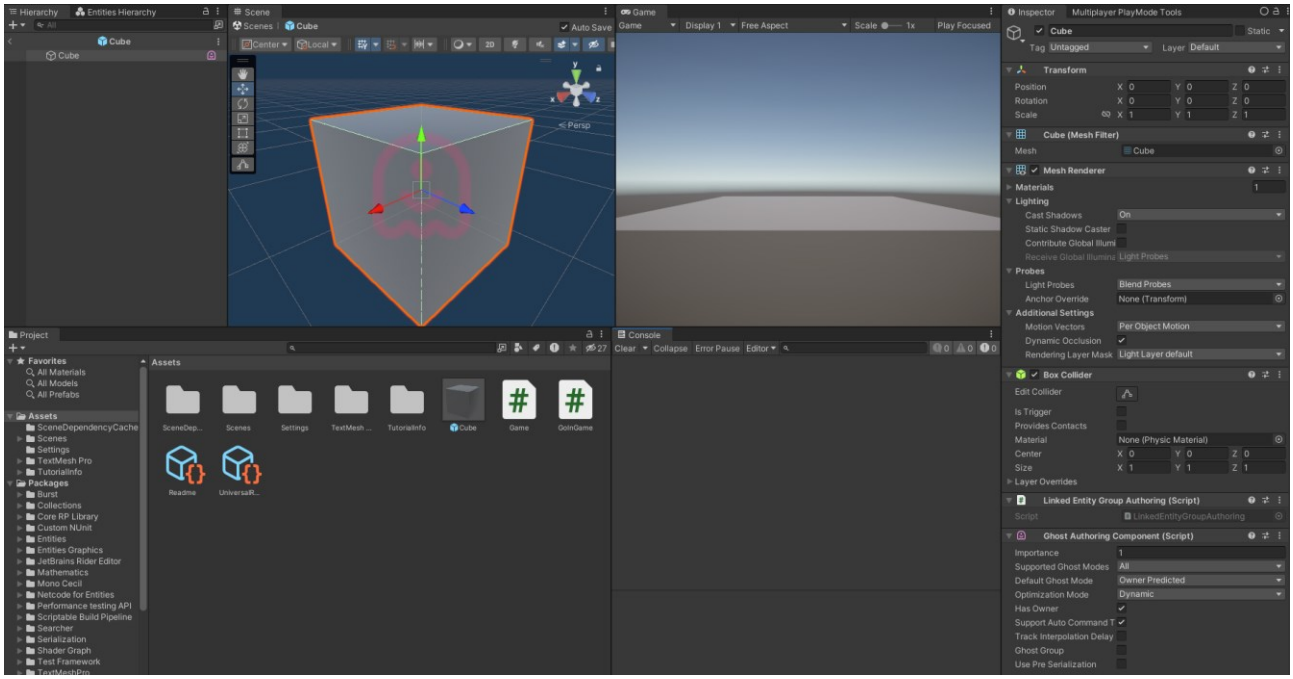
ClientServerBootstrap.cs-tiedoston Initialize-metodi suorittaa CreateWorld-metodin, joka luo pelaaja kohtaisen simulaation yhteyden tyypin perusteella (Kuva 7).

```
        public virtual bool Initialize(string defaultWorldName)
        {
            CreateDefaultClientServerWorlds();
            return true;
        }
        protected virtual void CreateDefaultClientServerWorlds()
        {
            var requestedPlayType = RequestedPlayType;
            if (requestedPlayType != PlayType.Client)
            {
                CreateServerWorld("ServerWorld");
            }

            if (requestedPlayType != PlayType.Server)
            {
                CreateClientWorld("ClientWorld");
            }
#if UNITY_EDITOR
            var requestedNumThinClients = RequestedNumThinClients;
            for (var i = 0; i < requestedNumThinClients; i++)
            {
                CreateThinClientWorld();
            }
#endif
        }
```

Kuva 7. ClientServerBootstrap.cs-tiedoston Initialize-metodin maailman luonti

5.3 Entities-projektin pelaajien lisäys peliin

Projektiin tehtiin uusi prefab-tiedosto Unityn omasta 3D-kappale kuutiosta, joka toimii yhdistäneiden pelaajien pelattavana hahmona. Prefab-tiedostoon lisättiin Entities-paketin oma komponentti nimeltä Ghost Authoring Component. Ghost Authoring Componentin oletus asetuksia muutettiin vaihtamalla omistaja radio valintalaatikko päälle ja omistaja ennustus Ghost moden valinnaksi. (Kuva 8.)



Kuva 8. Pelaajalle tehty prefab-tiedosto Unity-editorissa

Prefab-tiedoston kuutioon lisättiin uusi CubeAuthoring.cs-tiedosto komponentiksi, jonka avulla Entities-paketti tunnistaa kappaleen synkronoitavaksi pelaajien välillä (Kuva 9).

```
2 references
public struct Cube : IComponentData
{
}

[DisallowMultipleComponent]
Unity Script (1 asset reference) | 2 references
public class CubeAuthoring : MonoBehaviour
{
    0 references
    class Baker : Baker<CubeAuthoring>
    {
        0 references
        public override void Bake(CubeAuthoring authoring)
        {
            Cube component = default(Cube);
            AddComponent(component);
        }
    }
}
```

Kuva 9. Prefab-kappaleeseen lisätty CubeAuthoring.cs-tiedosto

Kaikki pelaajien väliset dynaamiset synkronoitavat kappaleet pitää suorittaa Baker-luokan Bake-metodin läpi, joka muuttaa Unityn authoring data-pelikappaleet Entities-paketin suoritusaika kappaleiksi. Synkronoitaville kappaleille pitää myös luoda IComponentDatan perivä rakenne, joka merkitsee kappaleet hallitsemattomiksi.

Projektiin tehtiin uusi tyhjä Spawner-kappale SharedData-kappaleen lapseksi. Spawner-kappaleeseen lisättiin komponentiksi uusi CubeSpawnerAuthoring.cs-tiedosto, joka synkronoi kappaleen CubeAuthoring.cs-tiedoston tapaan ja pitää sisällään rakenteen lisättäville pelaajien entity-kappaleille (Kuva 10).

```
7 references
public struct CubeSpawner : IComponentData
{
    public Entity Cube;
}

[DisallowMultipleComponent]
Unity Script (1 asset reference) | 2 references
public class CubeSpawnerAuthoring : MonoBehaviour
{
    public GameObject Cube;

    0 references
    class Baker : Baker<CubeSpawnerAuthoring>
    {
        0 references
        public override void Bake(CubeSpawnerAuthoring authoring)
        {
            CubeSpawner component = default(CubeSpawner);
            component.Cube = GetEntity(authoring.Cube);
            AddComponent(component);
        }
    }
}
```

Kuva 10. CubeSpawnerAuthoring.cs-tiedosto, jonka avulla pelaajille luodaan pelattavat kappaleet

Projektiin lisättiin GoInGame.cs-tiedosto, joka lähettää RPC-pyynnön pelaajan yhdistäessä palvelimelle datan synkronoinnin aloittamiseksi. Palvelimelta vastauksen saatua koodi poistaa RPC-pyynnön palvelimelta ja luo yhdistäneelle pelaajalle pelattavan kappaleen aikaisemmin luodun Spawner-kappaleen rakenteen osoittamasta prefab-tiedostosta. (Kuva 11.)

```
[BurstCompile]
[WorldSystemFilter(WorldSystemFilterFlags.ServerSimulation)]
1 reference
public partial struct GoInGameServerSystem : ISystem
{
    private ComponentLookup<NetworkId> networkIdFromEntity;

    [BurstCompile]
    0 references
    public void OnCreate(ref SystemState state)
    {
        state.RequireForUpdate<CubeSpawner>();
        var builder = new EntityQueryBuilder(Allocator.Temp)
            .WithAll<GoInGameRequest>()
            .WithAll<ReceiveRpcCommandRequest>();
        state.RequireForUpdate(state.GetEntityQuery(builder));
        networkIdFromEntity = state.GetComponentLookup<NetworkId>(true);
    }

    [BurstCompile]
    0 references
    public void OnUpdate(ref SystemState state)
    {
        var prefab = SystemAPI.GetSingleton<CubeSpawner>().Cube;
        state.EntityManager.GetName(prefab, out var prefabName);
        var worldName = state.WorldUnmanaged.Name;

        var commandBuffer = new EntityCommandBuffer(Allocator.Temp);
        networkIdFromEntity.Update(ref state);

        foreach (var (reqSrc, reqEntity) in SystemAPI.Query<RefRO<ReceiveRpcCommandRequest>>().WithAll<GoInGameRequest>().WithEntityAccess())
        {
            commandBuffer.AddComponent<NetworkStreamInGame>(reqSrc.ValueRO.SourceConnection);
            var networkId = networkIdFromEntity[reqSrc.ValueRO.SourceConnection];
            var player = commandBuffer.Instantiate(prefab);
            commandBuffer.SetComponent(player, new GhostOwner { NetworkId = networkId.Value});
            commandBuffer.AppendToBuffer(reqSrc.ValueRO.SourceConnection, new LinkedEntityGroup{Value = player});
            commandBuffer.DestroyEntity(reqEntity);
        }
        commandBuffer.Playback(state.EntityManager);
    }
}
```

Kuva 11. GoInGame.cs-tiedosto, joka huolehtii RPC-pyynnöistä ja pelattavien kappaleiden luomisesta yhteyksille

GoInGame.cs-tiedoston perittävä ISystem-luokka tekee koodista hallitsemattoman järjestelmän, minkä OnCreate-, OnUpdate- ja OnDestroy-metodit voidaan Burst-koostaa. BurstCompile-merkit merkitsevät tulevan koodin Burst-kääntäjän koostettavaksi korkeasti optimoiduksi natiivikoodiksi. WorldSystemFilter-merkit merkitsevät tulevan koodin suoritettavaksi suodatetuissa simulaatioissa. EntityManager-luokka on sovellusliittymä, jolla voi hallita, lukea ja päivittää entities-kappaleita. CommandBuffer-luokka laajentaa Unityn URP-systeemiä mukautetusti, johon voidaan lisätä jonoksi renderöitäviä komponentteja.

5.4 Entities-projektin pelaajien liikkeen tekeminen

Pelaajien pelattavaan prefab-kappaleeseen lisättiin komponentiksi uusi CubeInputAuthoring.cs-tiedosto, jonka avulla saatiin synkronoitava syötedata pelaaja hahmoihin kiinni (Kuva 12).

```
[GhostComponent(PrefabType=GhostPrefabType.AllPredicted)]
– references
public struct CubeInput : IInputComponentData
{
    public int Horizontal;
    public int Vertical;
}

[DisallowMultipleComponent]
Unity Script (1 asset reference) | – references
public class CubeInputAuthoring : MonoBehaviour
{
    – references
    class Baking : Unity.Entities.Baker<CubeInputAuthoring>
    {
        – references
        public override void Bake(CubeInputAuthoring authoring)
        {
            AddComponent<CubeInput>();
        }
    }
}
```

Kuva 12. Prefab-kappaleeseen lisätty CubeInputAuthoring.cs-tiedosto, jossa on rakenne syötedataa varten

CubeInputAuthoring.cs-tiedoston GhostComponent-merkit asettavat koodin kaikille ghost-instanseille ja määrittelevät missä niissä koodi on sallittu. Perittävä IInputComponentData-luokka on datakomponentti pelaajien syötteen varastoimiseksi, joka säilöö syötteen synkronoituun puskuriin asiakkaan ja palvelimen väliin.

CubeInputAuthoring.cs-tiedostoon lisättiin uusi OnUpdate-metodi päivittymään pelitilan kanssa, joka säilöö Unity-moottorin saamat pelaajien syötteet pelaajien hahmoihin kiinni (Kuva 13).

```
[UpdateInGroup(typeof(GhostInputSystemGroup))]
– references
public partial struct SampleCubeInput : ISystem
{
    – references
    public void OnUpdate(ref SystemState state)
    {
        bool left = UnityEngine.Input.GetKey("left");
        bool right = UnityEngine.Input.GetKey("right");
        bool down = UnityEngine.Input.GetKey("down");
        bool up = UnityEngine.Input.GetKey("up");

        foreach (var playerInput in SystemAPI.Query<RefRW<CubeInput>>().WithAll<GhostOwnerIsLocal>())
        {
            playerInput.ValueRW = default;
            if (left)
                playerInput.ValueRW.Horizontal -= 1;
            if (right)
                playerInput.ValueRW.Horizontal += 1;
            if (down)
                playerInput.ValueRW.Vertical -= 1;
            if (up)
                playerInput.ValueRW.Vertical += 1;
        }
    }
}
```

Kuva 13. CubeInputAuthoring.cs-tiedostoon lisätty OnUpdate-metodi, joka seuraa pelaajien syötteitä Unity-pelimoottorin puolelta

CubeInputAuthoring.cs-tiedoston UpdateInGroup-merkkien avulla määritellään Entities-paketin lisäämiä päivitysjärjestysryhmiä, joiden avulla voi ohjelmoida koodin päivittymään framen eri vaiheissa. SystemApi-luokka tarjoaa erilaisia tapoja päästä käsiksi projektin entities-kappaleiden dataan.

Projektiin lisättiin CubeMovementSystem.cs-tiedosto, joka pelitilan päivittymisen yhteydessä lukee pelaajiin säilöttyjä syötteitä ja liikuttaa pelaajakappaleita syötteiden mukaisesti (Kuva 14).

```
[UpdateInGroup(typeof(PredictedSimulationSystemGroup))]
[BurstCompile]
1 reference
public partial struct CubeMovementSystem : ISystem
{
    [BurstCompile]
    0 references
    public void OnUpdate(ref SystemState state)
    {
        var speed = SystemAPI.Time.DeltaTime * 4;
        var spin = quaternion.RotateY(SystemAPI.Time.DeltaTime * math.PI);
        var negativeSpin = Quaternion.Inverse(spin);

        foreach (var (input, trans) in SystemAPI.Query<RefRO<CubeInput>, RefRW<LocalTransform>>().WithAll<Simulate>())
        {
            var moveInput = new float2(input.ValueRO.Horizontal, input.ValueRO.Vertical);
            moveInput = math.normalizesafe(moveInput) * speed;
            trans.ValueRW.Position += ((moveInput.y * 1.8f) * trans.ValueRW.Forward());
            if (moveInput.x > 0)
            {
                trans.ValueRW.Rotation = math.mul(spin, trans.ValueRO.Rotation);
            }
            else if (moveInput.x < 0)
            {
                trans.ValueRW.Rotation = math.mul(negativeSpin, trans.ValueRO.Rotation);
            }
        }
    }
}
```

Kuva 14. CubeMovementSystem.cs-tiedosto, joka liikuttaa pelaajakappaleita pelitilan päivittyessä

Projektiin lisättiin CameraMovement.cs-tiedosto, joka ottaa viimeisimmän liittyneen pelaajan pelin kameran kohteeksi ja liikkuu tämän koordinaattien kanssa (Kuva 15).

```
[UpdateInGroup(typeof(LateSimulationSystemGroup))]
1 reference
public partial struct CameraMovement : ISystem
{
    Entity target;

    0 references
    public void OnUpdate(ref SystemState state)
    {
        if (target == Entity.Null)
        {
            var cubeQuery = SystemAPI.QueryBuilder().WithAll<Cube>().Build();
            var cubes = cubeQuery.ToEntityArray(Allocator.Temp);
            if (cubes.Length == 0)
            {
                return;
            }
            target = cubes[cubes.Length - 1];
            GoalSingleton.Instance.player = target.Index;
        }

        var cameraTransform = CameraSingleton.Instance.transform;
        var cubeTransform = SystemAPI.GetComponent<LocalToWorld>(target);
        cameraTransform.position = cubeTransform.Position;
        cameraTransform.position -= 2.0f * (Vector3)cubeTransform.Forward;
        cameraTransform.position += new Vector3(0, 0.6f, 0);
        cameraTransform.LookAt(new Vector3(cubeTransform.Position.x, cubeTransform.Position.y + 0.35f, cubeTransform.Position.z) );

        var cubeQuery2 = SystemAPI.QueryBuilder().WithAll<Cube>().Build();
        var cubes2 = cubeQuery2.ToEntityArray(Allocator.Temp);
        foreach (var cube in cubes2)
        {
            var cubeTrans = SystemAPI.GetComponent<LocalToWorld>(cube);
            if (cubeTrans.Position.x > -0.6f && cubeTrans.Position.x < 0.6f && cubeTrans.Position.z > -0.6f && cubeTrans.Position.z < -0.5f)
            {
                GoalSingleton.Instance.stopTimer(cube.Index);
            }
        }
    }
}
```

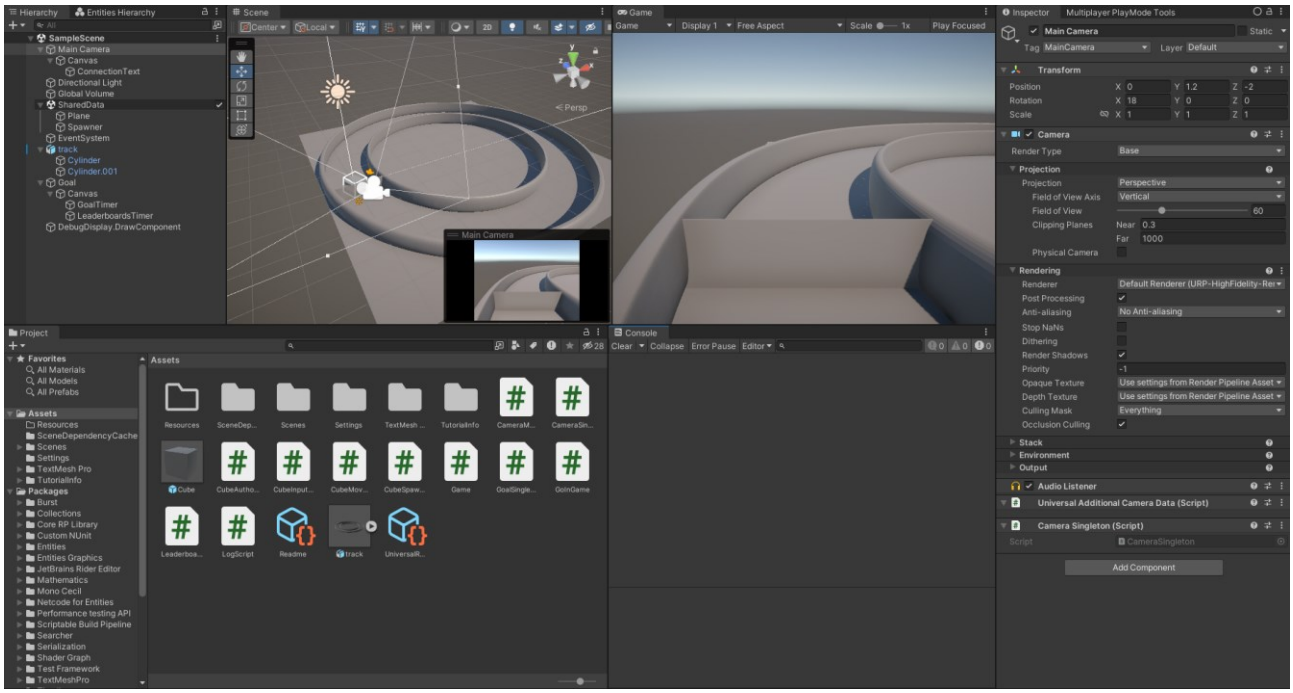
Kuva 15. CameraMovement.cs-tiedosto, joka huolehtii pelaajien kameran kohteesta ja liikkeestä

Projektiin lisättiin myös CameraSingleton.cs-, LeaderboardsSingleton.cs, GoalSingleton.cs- ja LogScript.cs-tiedostot viittaamaan paikallisiin pelimaailman kappaleisiin ja tuomaan tekstiä pelaajien ruuduille. Maailmaan lisättiin myös kahdesta rinkulasta koostuva rata.

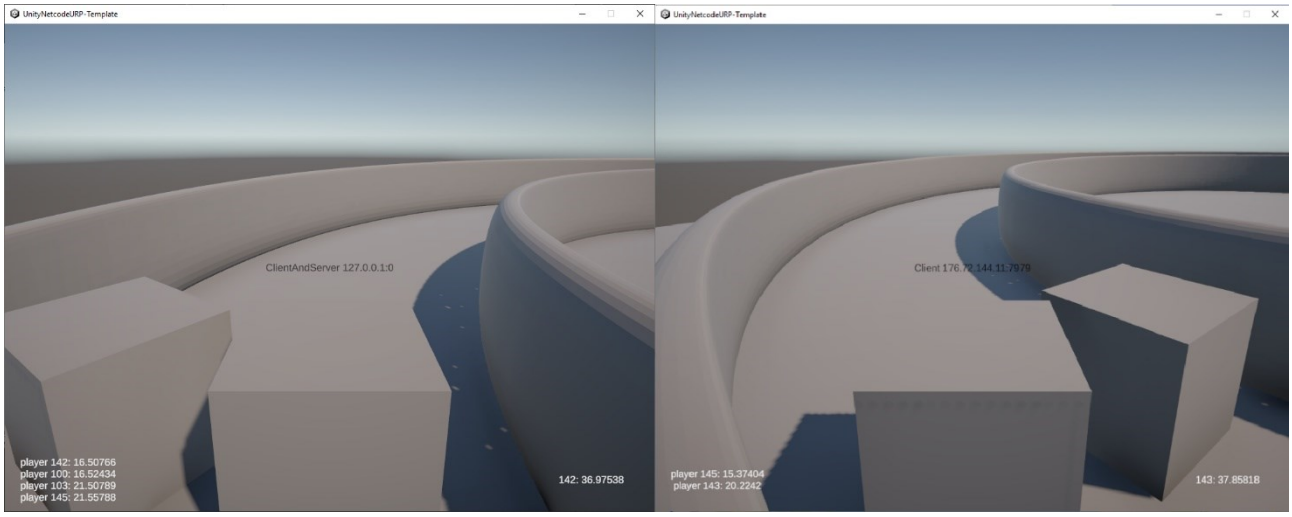
5.5 Entities-projektin lopputulos

Projektin toiminnallisuuden valmistuttua projekti piti buildata kahdelle eri netcode client targetille, clientille ja clientAndServerille. Buildaamisen jälkeen piti avata projektin koodissa asetettu portti reitittimestä, että ulkoiseen IP-osoitteeseen pystyi liittymään toisesta verkosta.

Valmiissa prototyypissä ilmeni puutteita rajallisesta ajasta aiheutuen, suurin näistä oli kappaleiden törmäyksien puute. Törmäyksien tekeminen prototyyppiin olisi vaatinut Unity Physics-paketin asentamista ja käyttöönottoa, joka olisi suurentanut tämän opinnäytetyön laajuutta entisestään (Kuva 16). Pienempiä puutteita oli useampia, kuten ClientAndServer-pelaajan useamman maaliajan rekisteröityminen pelaajien saapuessa maaliin, pelaajien eri numerot päätelaitteissa ja pelaajien paikallinen aika jaetun palvelinajan sijaan (Kuva 17).



Kuva 16. Valmis prototyyppi Unity-editorissa



Kuva 17. Buildattu projekti pelissä kahdella päätelaitteella eri verkoista

6 Photon Quantum prototyypin toteutus

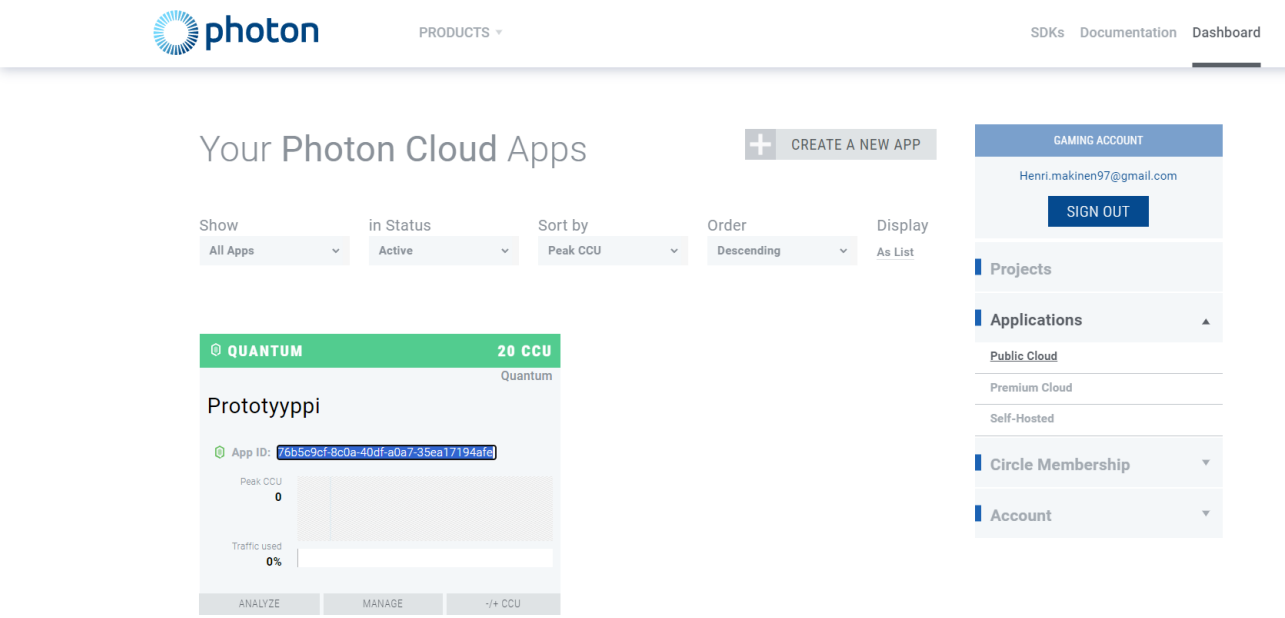
Unity Photon Quantum-verkkomoninpeliprototyypin toteutus tehtiin Photon Enginen-sivustolta ladatun Quantum-ohjelmistokehityspaketin sisältämään tyhjään projektipohjaan. Projektipohjaan toteutettiin Photon Enginen Quantum 100-dokumentaatioiden mukaisesti alustavaa sisältöä, jonka päälle jatkettiin tarvittavan toiminnallisuuden tekemistä itsenäisesti (Overview n.d.).

Quantum on ohjelmistokehityspaketin lisäksi myös pelimoottori, joten ohjelmoiminen tehtiin Unity-editorin sisäisten tiedostojen lisäksi Quantum-pelimoottorin koodiin, jota opinnäytetyössä kutsutaan jatkossa Quantum-koodiksi. Quantum-koodi piti jokaisen muutoksen jälkeen buildata, sillä Quantum-koodi oli erillinen koodikokoelma Quantum-pelimoottorin sisäisestä Unity-projektista. Quantum-ohjelmistokehityspaketti sisälsi valmiiksi kaikki prototyypin toiminnallisuuden tarvitsemat paketit, joten erillisiä paketteja ei tarvinnut lisätä Unityn omalla package managerilla.

6.1 Quantum-projektin pohja

Unity Photon Quantum-prototyyppiprojekti aloitettiin lataamalla uusin Quantum-ohjelmistokehityspaketti Photon Enginen-sivustolta. Uuden Unity-projektin tekemisen sijaan lisättiin olemassa oleva tyhjä projekti ladatun paketin sisältämistä tiedostoista.

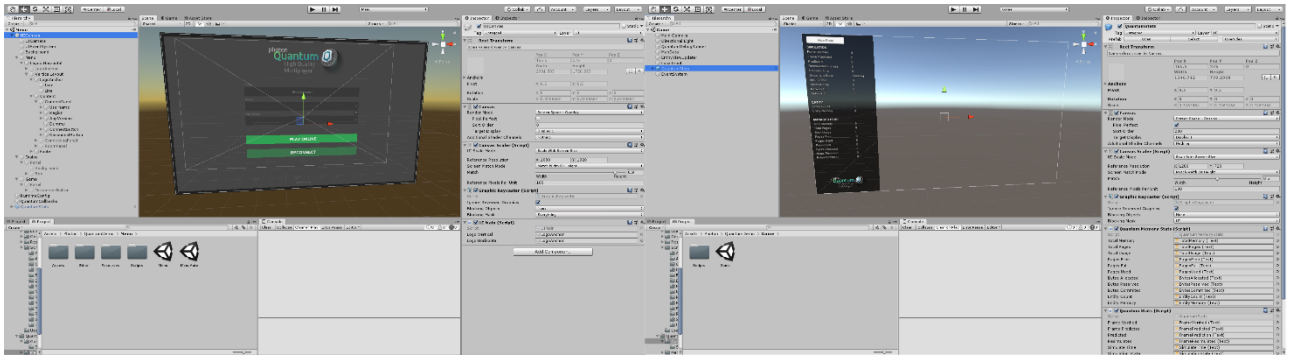
Lisättyyn tyhjään Quantum-projektiin piti aktivoida verkon yli pelaaminen Photonin palvelimien kautta, joka tehtiin lisäämällä Photon Enginen tarjoama AppID projektin sisällä olevaan PhotonServerSettings.asset-tiedostoon. Photon Enginen AppID:n sai rekisteröitymällä ja kirjautumalla Photon Enginen-sivustolle. Photon Engine-sivuston dashboardissa piti luoda uusi applikaatio Quantum-ohjelmistokehityspaketille, jonka tehtyä projektiin lisättävän AppID:n sai kopioitua dashboardin päänäkymästä (Kuva 18).



Kuva 18. Quantum-projektin AppID Photon Engine-sivuston dashboardissa

6.2 Quantum-projektin maailman alustus

Quantumin tyhjä projektipohja sisälsi valmiiksi kaksi tavallisesti pelinkehityksessä käytettävää kohtausta, valikko- ja pelikohtaukset. Valikkokohtaus koostui valmiista pelaajien yhdistämisen ja pelialuan käyttöliittymästä, kun taas pelikohtaus sisälsi tyhjän pelimaailman valmiilla moninpeli hallintakomponenteilla ja yhteystilastoilla (Kuva 19).

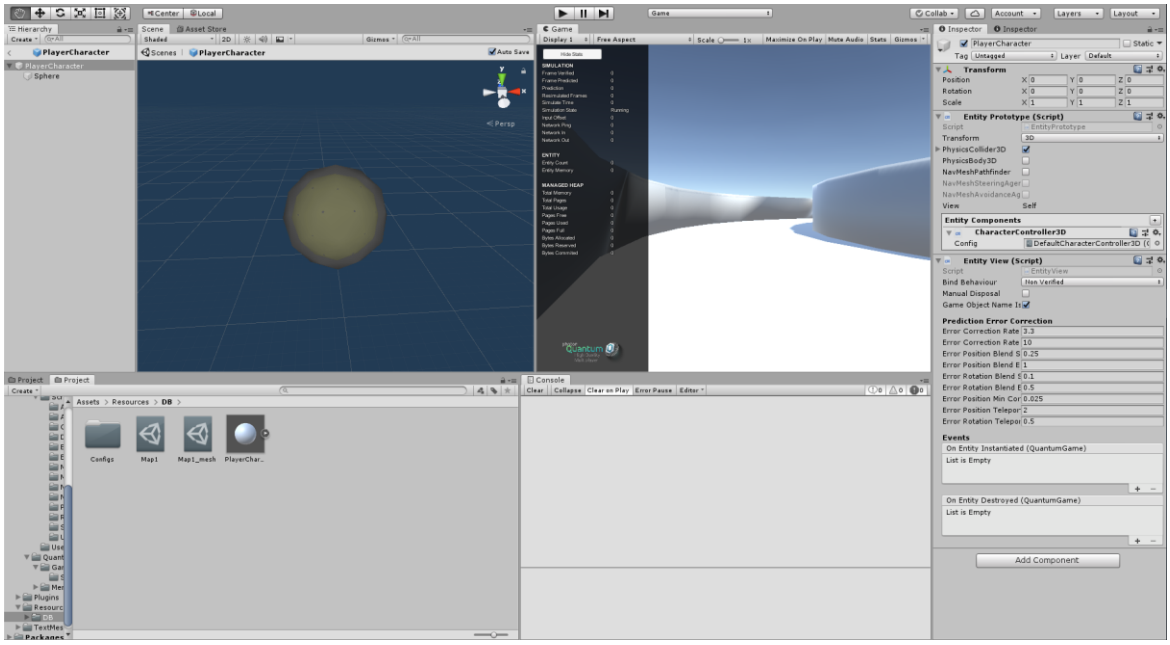


Kuva 19. Quantumin tyhjän projektipohjan Menu-scene ja Game-scene Unity-editorissa

Projektin tyhjään pelikohtaukseen lisättiin Quantumin oma static box collider-kappale kentän maatasoksi. Maatason päälle tuotiin Entities-projektissa käytetty rinkularata FBX-tiedostona, joka lisättiin kohtaukseen asettamalla se maataso-kappaleen meshiksi. Quantumin kaikki omat kappaleet sisältävät valmiiksi pelaajien välisen deterministisen synkronoinnin.

6.3 Quantum-projektin pelaajien liikkeen teko

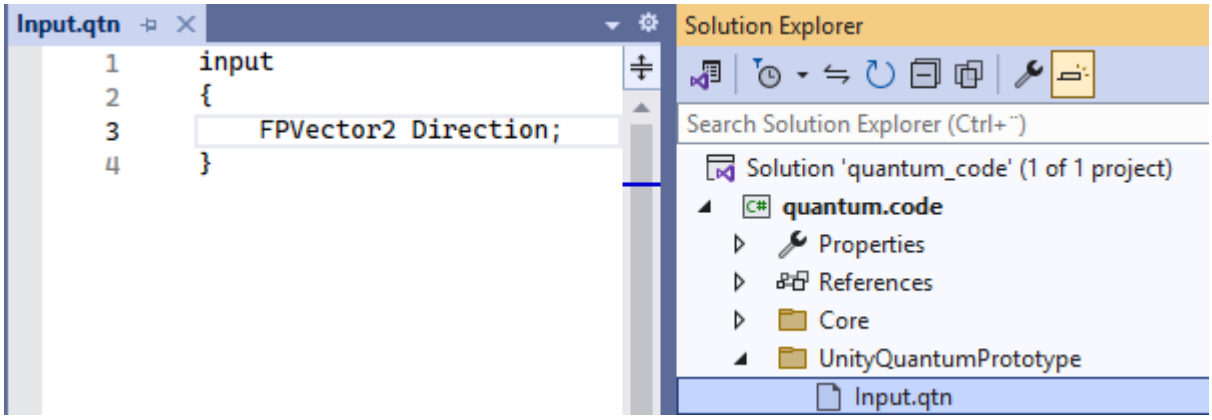
Projektiin luotiin uusi PlayerCharacter.prefab-tiedosto Quantumin omasta Character Controller Entity-kappaleesta, joka toimii yhdistäneiden pelaajien pelattavana hahmona (Kuva 20).



Kuva 20. PlayerCharacter.prefab-tiedosto avattuna pelikohtauksessa

Quantumin omassa character controller entity-kappaleessa on komponenttina EntityPrototype.cs-tiedosto, joka erottaa serialisoidun entity-kappaleen komponenttirakenteen ja tiedot toisistaan. Quantum-kappaleen komponenttina on myös EntityView.cs-tiedosto, joka hallitsee pelinäkymän puoleista koodia ja logiikkaa. Kappaleessa on myös valmiiksi viittaus DefaultCharacterController3D.asset-tiedostoon, josta voi muuttaa hahmon liikkumisen ominaisuuksia.

Projektin Quantum-koodin puolelle lisättiin uusi Input.qtn-tiedosto, joka määrittää verkon yli lähetettävän syötetiedon muuttujatyypin (kuva 21).



Kuva 21. Quantum-koodin puolelle lisätty Input.qtn-tiedosto

QTN-tiedosto on Quantumin oma tiedostoformaatti, joka käyttää Quantumin verkkotunnuskohtaista kieltä. Quantumin verkkotunnuskohtaista kieltä käytetään määrittelemään entity-kappaleiden komponenttitietoja ja syötetietoja. FP-kirjaimet Vector2-muuttujatyypin edessä tarkoittavat kiintopistemuuttujaa, joka korvaa kaikki float- ja double-muuttujat Quantumissa varmistaakseen alustojen välisen deterministisyyden.

Projektiin lisättiin MovementSystem.cs-tiedosto, joka päivittää suodatettujen entity-kappaleiden sijaintia ja suuntaa Input.qtn-tiedoston syötteiden mukaisesti (Kuva 22).

```
public unsafe class MovementSystem : SystemMainThreadFilter<MovementSystem.Filter>
{
    2 references
    public struct Filter
    {
        public EntityRef Entity;
        public CharacterController3D* CharacterController;
    }

    2 references
    public override void Update(Frame f, ref Filter filter)
    {
        Input input = default;
        if (f.Unsafe.TryGetPointer(filter.Entity, out PlayerLink* playerLink))
        {
            input = *f.GetPlayerInput(playerLink->Player);
        }

        if (f.Unsafe.TryGetPointer<Transform3D>(filter.Entity, out var transform))
        {
            var moveVector = FPVector3.Scale(new FPVector3(10, 10, 10), transform->Forward);
            if (input.Direction.Y > 0)
            {
                filter.CharacterController->Move(f, filter.Entity, moveVector);
            }
            else if (input.Direction.Y < 0)
            {
                filter.CharacterController->Move(f, filter.Entity, moveVector * -1);
            }
            else
            {
                filter.CharacterController->Move(f, filter.Entity, default);
            }

            if (input.Direction.X > 0)
            {
                transform->Rotation *= FPQuaternion.Euler(0, 3, 0);
            }
            else if (input.Direction.X < 0)
            {
                transform->Rotation *= FPQuaternion.Euler(0, -3, 0);
            }
        }
    }
}
```

Kuva 22. MovementSystem.cs-tiedosto, joka liikuttaa pelaajia Input.qtn-tiedoston syötteiden mukaisesti

MovementSystem-luokka on merkitty unsafe-koodiksi, joka tarkoittaa, että .NET työkalut eivät voi tarkistaa koodia turvalliseksi. Turvallinen koodi ei voi varata raakaa muistia, eikä sillä ole suoraa pääsyä muistiin osoittimien avulla, vaan turvallinen koodi tekee hallittuja kappaleita muistiin. Quantumin avain korkeaan suorituskykyyn on osoitinpohjainen koodi yhdistettynä ECS-muistimalliin. Tiedoston perittävä SystemMainThreadFilter-luokka käy läpi kaikki olemassa olevat entity-kappaleet asetetulla komponentti suodatuksella. Tiedoston Frame-luokka on välttämätön Quantumin deterministisyydelle, sillä se validoi pelaajien syötteet palvelimella ja huolehtii framejen palautuksesta ja uudelleen simuloimisesta.

Projektipohjan mukana tulleeseen LocalInput.cs-tiedostoon lisättiin kuuntelija lähettämään Unity-pelimoottorin saamat syötteet Quantum-pelimoottorille (Kuva 23).

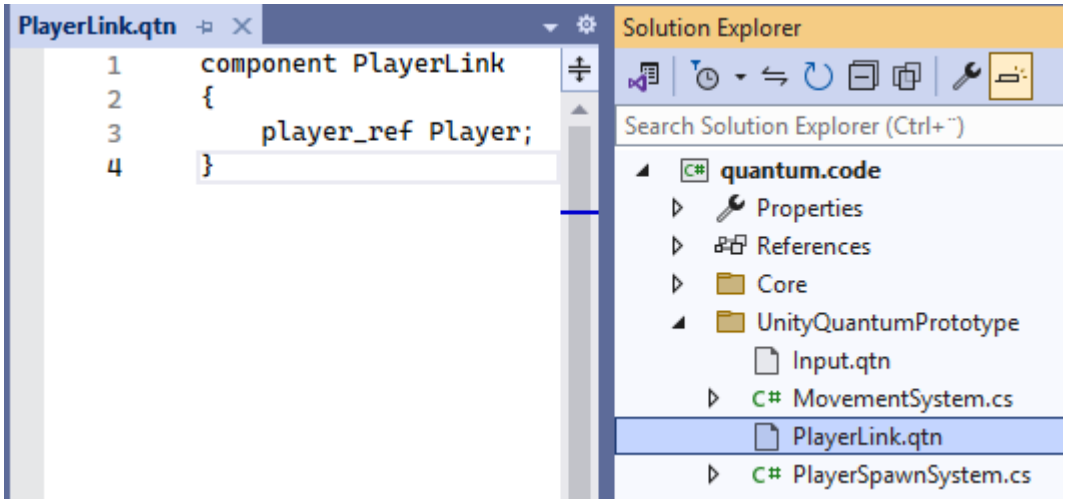
```
Unity Script | 0 references
public class LocalInput : MonoBehaviour
{
    Unity Message | 0 references
    private void Start()
    {
        QuantumCallback.Subscribe(this, (CallbackPollInput callback) => PollInput(callback));
    }

    1 reference
    public void PollInput(CallbackPollInput callback)
    {
        Quantum.Input input = new Quantum.Input();

        var x = UnityEngine.Input.GetAxis("Horizontal");
        var y = UnityEngine.Input.GetAxis("Vertical");

        input.Direction = new Vector2(x, y).ToFPVector2();

        callback.SetInput(input, DeterministicInputFlags.Repeatable);
    }
}
```

Kuva 23. LocalInput.cs-tiedosto, joka lähettää Unity-pelimoottorin syötteet Quantum-pelimoottorille

QuantumCallback-tapahtumat ovat Quantumin omia tapahtumia, jotka laukeavat Quantum-pelimoottorin sisältä. CallbackPollInput-tapahtumaa kutsutaan, kun simulaatio tekee paikallisen syötekyselyn. DeterministicInputFlags.Repeatable-tyyppi merkitsee syötedatan toistettavaksi palvelimen ja kaikkien yhdistäneiden pelaajien tulevissa tickeissä.

6.4 Quantum-projektin pelaajien lisäys peliin

Projektin Quantum-koodin puolelle lisättiin PlayerLink.qtn-tiedosto, jota käytetään yhdistämään yhdistäneiden pelaajien uniikit PlayerRef ID:t pelaajien entity-kappaleisiin (Kuva 24).



Kuva 24. Quantum-koodin puolelle lisätty PlayerLink.qtn-tiedosto

Quantum-pelimoottorilla on käsitys yhdistäneistä pelaajista, mutta se ei tiedä automaattisesti yhteyksien pelattavista entity-kappaleista. PlayerLink-komponentti on Quantumin oma tapa yhdistää pelin entity-kappaleet yhteyksille.

Quantum-koodin puolen RuntimePlayer.User.cs-tiedostoon lisättiin AssetRefEntityPrototype-luokkainen julkinen muuttuja, joka toimii pelaajan prefab-kappaletta vastaavana Quantum-kappaleena (Kuva 25).

```
partial class RuntimePlayer
{
    public AssetRefEntityPrototype CharacterPrototype;

    2 references
    partial void SerializeUserData(BitStream stream)
    {
        stream.Serialize(ref CharacterPrototype);
    }
}
```

Kuva 25. Quantum-koodin RuntimePlayer.User.cs-tiedostoon lisätty julkinen muuttuja

RuntimePlayer-luokka säilyttää pelaaja kohtaista tietoa, jonka avulla jokainen pelaaja pystyy lähettämään tietoa yhteyden simulaatiolle. Kaikki RuntimePlayer-luokkaan lisätty tieto pitää serialisoida

SerializeUserData-metodin kautta. BitStream-luokka tarjoaa serialisointi-metodeja kaikille tavallisille tyypeille.

Projektin Quantum-koodin puolelle lisättiin PlayerSpawnSystem.cs-tiedosto, joka PlayerDataSet-tapahtuman yhteydessä luo frameen pelaajalle entity-kappaleen ja lisää siihen PlayerLink-luokan (Kuva 26).

```
namespace Quantum.Game
{
    1 reference
    unsafe class PlayerSpawnSystem : SystemSignalsOnly, ISignalOnPlayerDataSet
    {
        2 references
        public void OnPlayerDataSet(Frame frame, PlayerRef player)
        {
            var data = frame.GetPlayerData(player);
            var prototype = frame.FindAsset<EntityPrototype>(data.CharacterPrototype.Id);
            var entity = frame.Create(prototype);

            var playerLink = new PlayerLink()
            {
                Player = player,
            };
            frame.Add(entity, playerLink);

            if (frame.Unsafe.TryGetPointer<Transform3D>(entity, out var transform))
            {
                transform->Position.X = (int)player;
            }
        }
    }
}
```

Kuva 26. Quantum-koodin puolelle lisätty PlayerSpawnSystem.cs-tiedosto, joka luo pelaajalle pelattavan entity-kappaleen

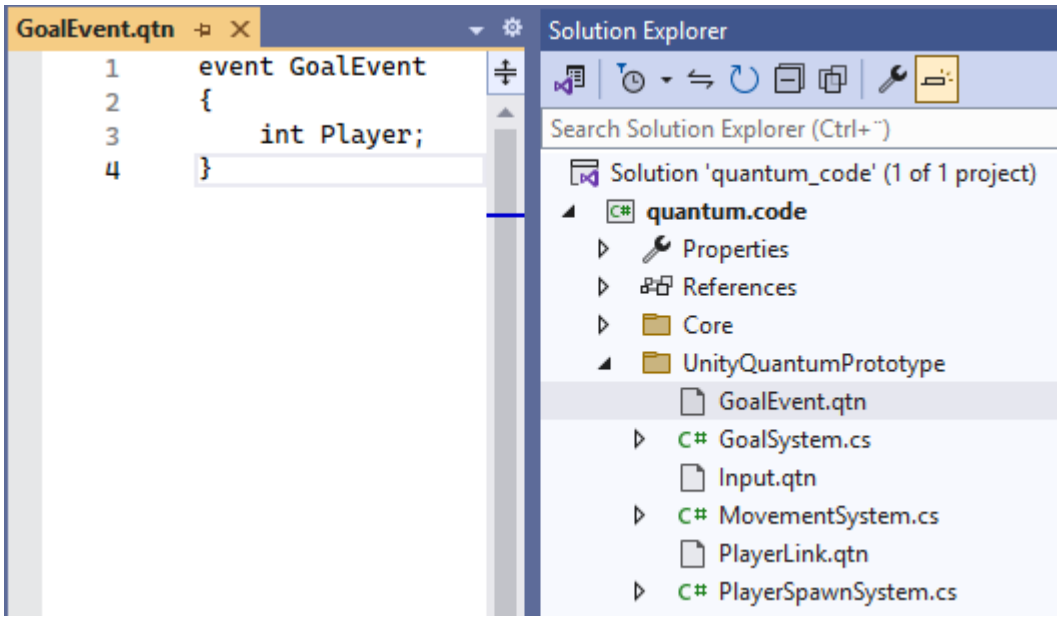
Quantum-signaalit ovat järjestelmän callback-tapahtumia. Tiedoston perittävä SystemSignalsOnly-luokka sallii koodin käsitellä signaaleja varaamatta muistia natiivitapahtumametodeille. Perittävä ISignalOnPlayerDataSet-luokka on rajapintaluokka, mikä mahdollistaa kyseisen tapahtuman vastaanottamisen.

Projektiin lisättiin CameraHandler.cs-tiedosto pelaajan prefab-kappaleen komponentiksi, joka päivittää prefab-kappaleen luonnin yhteydessä asetettua kameran sijaintia paikallisesti (Kuva 27).

```
public class CameraHandler : MonoBehaviour
{
    [SerializeField] EntityView entityView;
    private Transform cameraTransform;

    0 references
    public void OnEntityInstantiated()
    {
        QuantumGame game = QuantumRunner.Default.Game;
        Frame frame = game.Frames.Verified;

        if (frame.TryGet(entityView.EntityRef, out PlayerLink playerLink))
        {
            if (game.PlayerIsLocal(playerLink.Player))
            {
                cameraTransform = GameObject.Find("Main Camera").transform;
            }
        }
    }

    Unity Message | 0 references
    private void Update()
    {
        if (cameraTransform != null)
        {
            cameraTransform.position = transform.position;
            cameraTransform.position -= 4.0f * (Vector3)transform.forward;
            cameraTransform.position += new Vector3(0, 2f, 0);
            cameraTransform.LookAt(new Vector3(transform.position.x, transform.position.y + 1f, transform.position.z));
        }
    }
}
```

Kuva 27. CameraHandler.cs-tiedosto, joka huolehtii pelaajien kameran sijainnista

Projektin Quantum-koodin puolelle lisättiin GoalEvent.qtn-tiedosto, jota käytetään maalitapahtumissa siirtämään maaliin törmänneiden entity-kappaleiden pelaajanumeroa eteenpäin (Kuva 28).



Kuva 28. Quantum-koodin puolelle lisätty GoalEvent.qtn-tiedosto

Projektin Quantum-koodin puolelle lisättiin GoalSystem.cs-tiedosto, joka lähettää maaliin törmänneen entity-kappaleen PlayerLink-luokan pelaajanumeron maalitapahtumalle (Kuva 29).

```
unsafe class GoalSystem : SystemSignalsOnly, ISignalOnTrigger3D
{
    2 references
    public void OnTrigger3D(Frame f, TriggerInfo3D info)
    {
        if (f.TryGet(info.Entity, out PlayerLink playerLink))
        {
            f.Events.GoalEvent(playerLink.Player._index);
        }
    }
}
```

Kuva 29. Quantum-koodin puolelle lisätty GoalSystem.cs-tiedosto, joka lähettää pelaajanumerot maalitapahtumalle

Projektiin lisättiin UIHandler.cs-tiedosto, joka lisää kuuntelijan maalitapahtumaan, päivittää pelaajien aikaa paikallisesti ja rekisteröi kaikkien pelaajien maalin ylitysajat paikallisiin UI-tekstielementteihin (Kuva 30).

```
public unsafe class UIHandler : MonoBehaviour
{
    [SerializeField] EntityView entityView;
    private TextMeshProUGUI leaderboardTextMesh;
    private TextMeshProUGUI timeTextMesh;
    private float currentTime;
    private List<int> wonPlayers;
    private static int thisPlayer;

    Unity Message | 0 references
    private void Awake()
    {
        QuantumEvent.Subscribe(listener: this, handler: (EventGoalEvent e) => AddToLeaderboard(e.Player));
        leaderboardTextMesh = GameObject.Find("Leaderboard").GetComponent<TextMeshProUGUI>();
        timeTextMesh = GameObject.Find("Time").GetComponent<TextMeshProUGUI>();
        timeTextMesh.fontSize = 18;
        wonPlayers = new List<int>();
    }

    0 references
    public void OnEntityInstantiated()
    {
        QuantumGame game = QuantumRunner.Default.Game;
        Frame frame = game.Frames.Verified;
        if (frame.TryGet(entityView.EntityRef, out PlayerLink playerLink))
        {
            if (game.PlayerIsLocal(playerLink.Player))
            {
                thisPlayer = playerLink.Player._index;
            }
        }
    }
```

Kuva 30. UIHandler.cs-tiedosto, joka huolehtii pelaajan käyttöliittymässä näytettävistä ajoista

Projektin Quantum-koodin puolen SystemSetup.cs-tiedostosta piti poistaa projektissa käyttämättömäksi jääneet järjestelmät käytöstä ja lisätä omat järjestelmät käyttöön (Kuva 31).

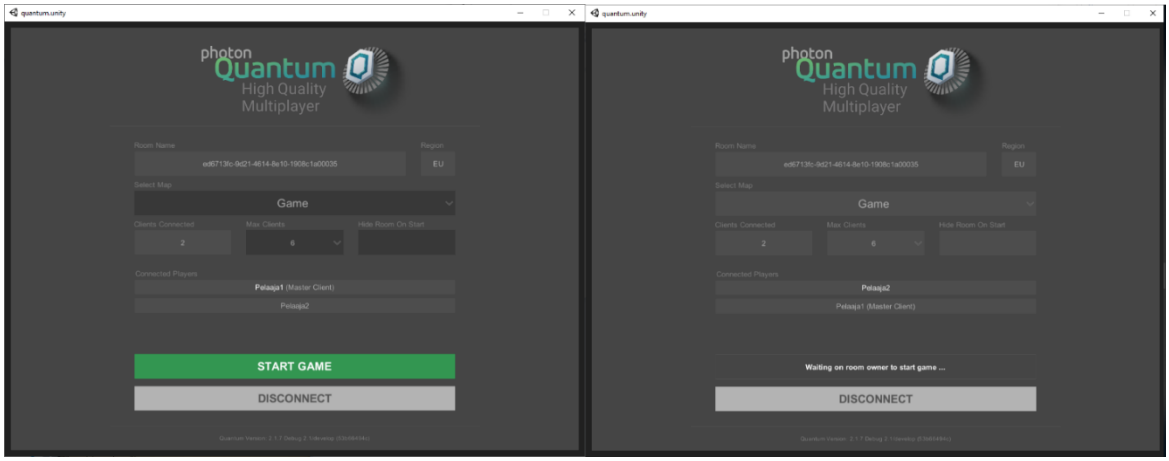
```
namespace Quantum {
    1 reference
    public static class SystemSetup {
        1 reference
        public static SystemBase[] CreateSystems(RuntimeConfig gameConfig, SimulationConfig simulationConfig) {
                return new SystemBase[] {
            // pre-defined core systems
            //new Core.CullingSystem2D(),
            new Core.CullingSystem3D(),

            //new Core.PhysicsSystem2D(),
            new Core.PhysicsSystem3D(),

            Core.DebugCommand.CreateSystem(),

            //new Core.NavigationSystem(),
            new Core.EntityPrototypeSystem(),
            new Core.PlayerConnectedSystem(),

            // user systems go here
            new MovementSystem(),
            new PlayerSpawnSystem(),
            new GoalSystem(),
          };
        }
    }
}
```

Kuva 31. Quantum-koodin puolen SystemSetup.cs-tiedosto, joka huolehtii käytössä olevista järjestelmistä

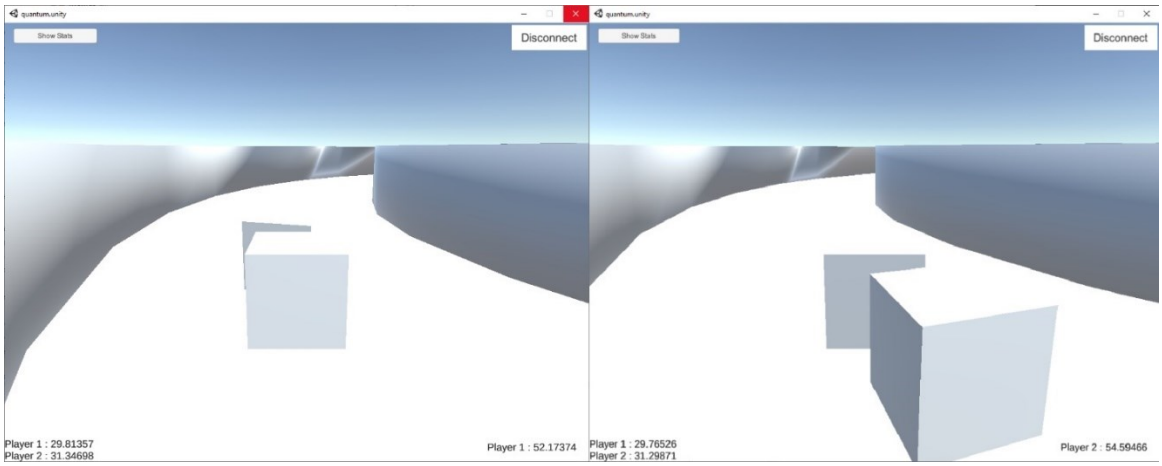
6.5 Quantum-projektin lopputulos

Projektin toiminnallisuuden valmistuttua projektiin piti buildata molemmat kohtaukset samaan pakettiin, sillä kohtaukset olivat suunniteltu toimimaan yhdessä Quantum-projektipohjan lataamisesta asti (Kuva 32). Buildaamisen jälkeen yhdistäminen samaan peliin eri verkoista toimi ilman erillisiä porttien availuja, sillä peliaula yhdistää jokaisen pelaajan Photonin palvelimille (Kuva 33).

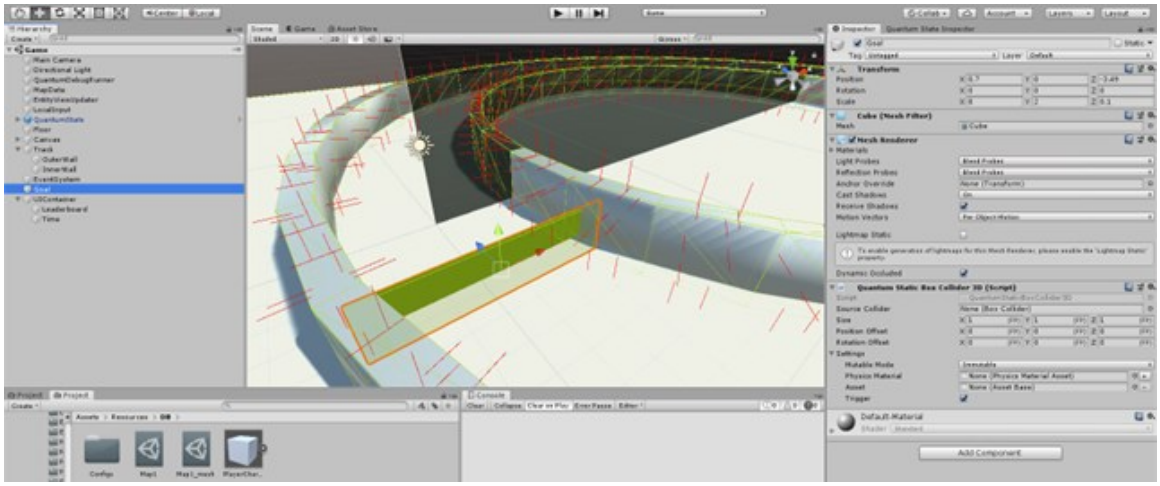
Valmiissa prototyypissä ilmeni paljon vähemmän puutteita rajallisesta ajasta aiheutuen, sillä valmiit deterministiset komponentit ja peliaula nopeuttivat prosessia huomattavasti (Kuva 34).



Kuva 32. 2 pelaajaa yhdistämisaulassa eri verkoista



Kuva 33. 2 pelaajaa pelissä eri verkoista

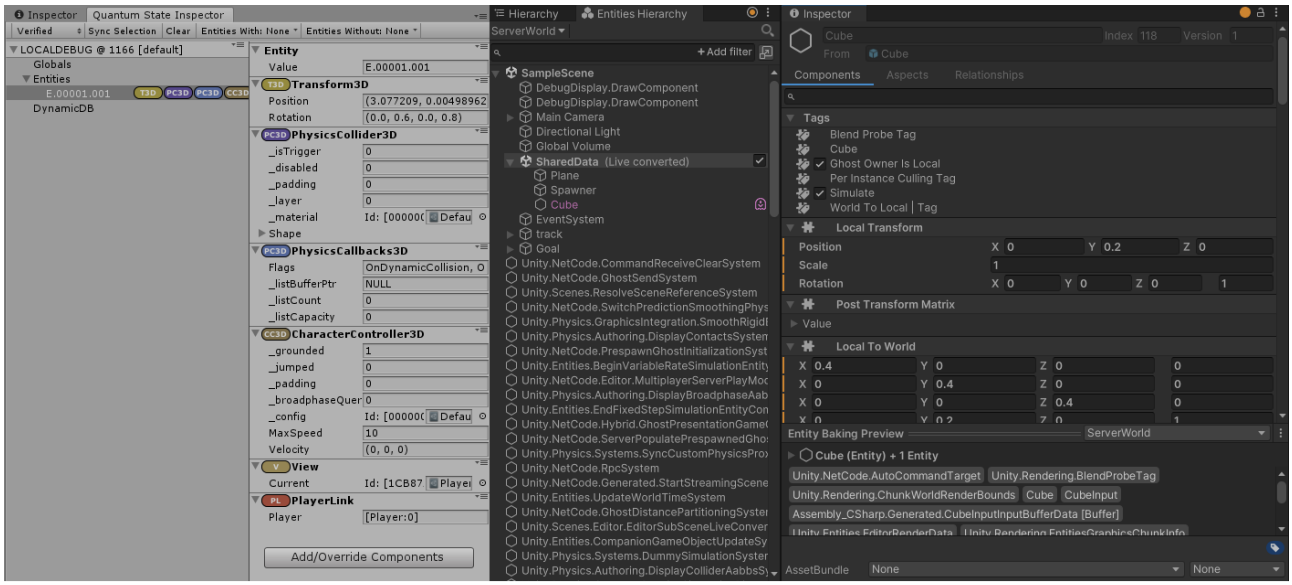


Kuva 34. Valmis projekti Unity-editorissa

7 Prototyyppien analyysi ja vertailu

Unity Netcode For Entities- ja Unity Photon Quantum-prototyyppien toteutuksien prosesseissa oli todella paljon erilaisuuksia, vaikka molemmat pohjautuivat ECS-ohjelmistoarkkitehtuuriin. Entities-prototyypissä kaikki pelaajien välille synkronoitavaksi tarkoitetut kappaleet piti suorittaa Baker-luokan läpi muuttaakseen ne authoring-tiedoiksi, kun taas Quantum-prototyypissä piti pelaajien välille synkronoitavaksi tarkoitetut kappaleet tehdä Quantumin omista valmiiksi deterministisistä kappaleista. Entities-prototyypin tietojen siirto entity-kappaleiden ja pelimaailmaan välillä oli paljon hankalampaa kuin Quantum-prototyypissä, sillä Quantumin valmiit järjestelmätapahtumat ja QTN-tiedostot helpottivat tietojenhallintaa entity-kappaleiden ja pelimaailman välillä.

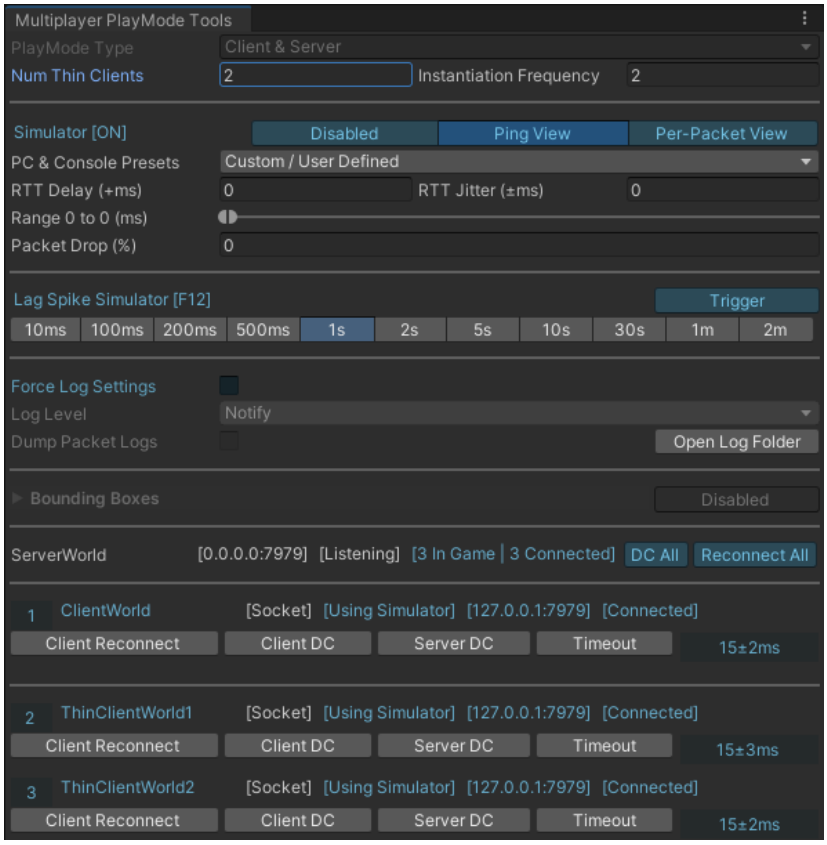
Molempien prototyyppien yhteydet ja tiedonsiirrot pelaajien välillä tuntuivat nopeilta, mutta prototyyppien verkon yli liikuteltavan tiedon määrä oli niin pieni, ettei siitä pystynyt päättelemään tehokkuuksia ilman erillisiä tiedonsiirto stressitestejä. Molemmissa prototyypeissä tuli asennetun paketin mukana entities inspector-työkalu, joka oli Quantum-paketissa helppokäyttöisempi, mutta näytti laajemmin tietoja Netcode For Entities-paketissa (Kuva 35).



Kuva 35. Molempien projektien entities inspector-työkalut Unity-editorissa

Entities-prototyypissä piti buildata kaksi erillistä versiota luomaan peli vertaisverkon isännälle ja yhdistämään pelaajat siihen, kun taas Quantum-prototyypissä yksi buildattu versio yhdisti kaikki

pelaajat asiakas-palvelin-mallisen pelin palvelimelle. Entities-prototyyppi yhdisti pelaajat vain asetettuun IP-osoitteeseen, kun taas Quantum-prototyypissä oli valmiit matchmaking- ja aulajärjestelmät yhdistämään pelaajat Photonin palvelimiin. Netcode For Entities-paketissa oli editorin ajonaikaiset yhteystyökalut moninpelin verkkotoimintojen testaamiseen, joihin kuului muun muassa kevyt asiakkaiden lisäys ja yhteysongelmasimulointityökaluja (Kuva 36).



Kuva 36. Netcode For Entities-paketin ajonaikaiset yhteystyökalut Unity-editorissa

8 Tulos ja pohdinta

Unity Netcode for Entities-prototyypissä oli paljon enemmän työtä ja perehtymistä suppeampaan toteutukseen, kuin Unity Photon Quantum-prototyypissä. Entities-paketilla työskentely riittävällä kokemuksella ja tietämyksellä olisi varmasti johdonmukaista ja nopeata, sillä kaikki pelaajien välille synkronoitavat entities-kappaleet suoritetaan aina saman baker-luokan läpi. Quantum-paketilla työskentely oli uudellekin työstäjälle helppoa ja tehokasta Quantumin valmiiden deterministisen kappaleiden ansiosta, sillä ne synkronoivat entities-kappaleet pelaajien välille automaattisesti.

Entities-kappaleiden ja pelimaailman välinen tiedonsiirto oli myös huomattavasti yksinkertaisempaa Quantumissa valmiiden järjestelmätapahtumien ja QTN-tiedostojen ansiosta.

Opinnäytetyön tavoitteena oli selvittää täysin deterministisen pelimoottorin Quantumin tuomia hyötyjä ja tehokkuuksia pelinkehityksessä käytännön prototyyppien kautta. Opinnäytetyössä toteutettiin ensimmäinen prototyyppi osaksi deterministisellä Netcode For Entities-ohjelmistokehityspaketilla ja toinen täysin deterministisellä Quantum-ohjelmistokehityspaketilla.

Molempien prototyyppien toteuttaminen oli aikaa vievää niiden ohjelmistokehityspaketteihin perehtymisen takia. Quantum-pakettiin perehtymisen aikana vastaan tuli selvästi enemmän helposti käyttöön otettavia ohjelmallisia korjauksia moninpelien verkko-ongelmiin, kuten framejen ennustus/peruutus-, asiakaspuolen ennustus- ja viiveen korvaustekniikoita. Perehtymisen jälkeen projektin ominaisuuksien toteuttaminen oli paljon nopeampaa ja tehokkaampaa täysin deterministisellä Quantum-ohjelmistokehityspaketilla, sillä ohjelmistokehitysympäristönä se oli paljon pidemmälle viety helposti muokattavien valmis kappaleiden ja käyttäjäystävällisyyden ansiosta. Deterministisyyden ylläpito Quantum-prototyypissä ei aiheuttanut yhtään lisätyötä järkevien QTN-tiedosto muuttujien ja valmiiksi pelaajien välille synkronoitujen kappaleiden ansiosta.

Opinnäytetyössä onnistuttiin toteuttamaan kaksi samanlaista verkkomoninpeliprototyyppiä kahdella eri ECS-ohjelmistoarkkitehtuuriin pohjautuvalla ohjelmistokehityspaketilla ilman edeltävää tietoa tai kokemusta paketeista. Netcode For Entities-prototyyppi toteutettiin yhden isännän vertaisverkon verkkorakenteella, kun taas Quantum-prototyyppi toteutettiin Quantumin valmiin yhdistämisaulan ansiosta asiakas-palvelin-mallisesti Photonin palvelimille. Prototyyppien toteutus onnistui hyvin, mutta Entities-prototyypin törmäyksien tekeminen epäonnistui, sillä ominaisuuden toteuttamiseksi olisi tarvinnut asentaa ja opetella erillinen ohjelmistokehityspaketti.

Opinnäytetyön prototyyppien toteutusmenetelmien luotettavuudessa on rajoituksia kaistanrasituksen ja yhteysongelmien testaamisen puutteen takia. Kummassakaan prototyypissä ei myöskään käytetty Quantumin sisäänrakennetun ennustus/peruutus lähestymistavan lisäksi mitään moninpelikokemusta parantavaa verkko-ohjelmointimenetelmää, sillä pienen pelaajien välisen tiedon liikkumisen takia kaikki yhteyden aiheuttamat rajoitukset olivat hyvin huomaamattomia. Tiedonsiirron stressitestejä ei näillä prototyypeillä pystynyt tekemään, sillä pelaajien välistä tietoa liikkui

liian vähän. Prototyyppien toteutuksien tuloksissa on puutteita, sillä kaikki opinnäytetyön vertailutulokset ovat subjektiivisia ja riippuvaisia pelkästään tekijän kokemuksesta.

Opinnäytetyössä toteutettujen prototyyppiprojektien vertailutuloksia voi hyödyntää käyttämällä aiheeseen liittyvää kokemusta ja järkevää vertailua muiden moninpelien kehittämisprosesseihin ohjelmistokehityspaketeilla. Opinnäytetyössä kehuttua Quantum-ohjelmistokehityspakettia pitäisi pystyä katsomaan objektiivisesti moninpelikehitysprojektin tarvittavien osa-alueiden kannalta ja pystyä suhteuttamaan Quantumin tuomat hyödyt ja tehokkuudet tämän hintapolitiikkaan.

Opinnäytetyössä esille tullut ECS-ohjelmistoarkkitehtuuri oli paljon odotettua hyödyllisempi ja helpommin ymmärrettävä. Täysin deterministinen Quantum-pelimoottori osoittautui erittäin tehokkaaksi ja helppokäyttöiseksi työkaluksi.

Lähteet

Amiga Flight Simulator II. 1988. Manuaali OldGamesDownload-sivustolla. Viitattu 02.12.2023. https://oldgamesdownload.com/wp-content/uploads/Flight_Simulator_II_Manual_Amiga_EN.pdf.

Andrea, H. N.d. Comparison of “peer-to-peer” vs “client-server” Network Models. Artikkeli Networks training-sivustolla. Viitattu 05.04.2024. https://www.networkstraining.com/peer-to-peer-vs-client-server-network/.

Axon, S. 2016. Unity at 10: For better—or worse—game development has never been easier. Artikkeli Ars Technica- sivustolla. Viitattu 07.01.2024. https://arstechnica.com/gaming/2016/09/unity-at-10-for-better-or-worse-game-development-has-never-been-easier/.

Bernier, Y. 2001. Latency Compensating Methods in Client/Server In-game Protocol Design and Optimization. Artikkeli Valve-sivustolla. Viitattu 03.12.2023. https://developer.valvesoftware.com/wiki/Latency_Compensating_Methods_in_Client/Server_In-game_Protocol_Design_and_Optimization#Lag_Compensation.

Build multiplayer games with Unity netcode. N.d. Sivu Unity-sivustolla. Viitattu 08.01.2024. https://unity.com/products/netcode.

China Mobile Games Market & 5-Year Forecast Report 2023. 2023. Reportaasi Nikopartners-sivustolla. Viitattu 20.11.2023. https://nikopartners.com/china-mobile-games/.

Clement, J. 2023. Number of games released on Steam 2004-2023. Tilasto artikkeli Statista-sivustolla. Viitattu 21.11.2023. https://www.statista.com/statistics/552623/number-games-released-steam/.

Client-Server. 2023. Artikkeli Heavy.AI-sivustolla. Viitattu 03.12.2023. https://www.heavy.ai/technical-glossary/client-server.

Client-Server Model. 2022. Artikkeli Geeks for geeks-sivustolla. Viitattu 05.04.2024. https://www.geeksforgeeks.org/client-server-model/.

Client-Server Network: Definition, Advantages, and Disadvantages. 2023. Artikkeli Zenarmor-sivustolla. Viitattu 05.04.2024. https://www.zenarmor.com/docs/network-basics/what-is-client-server-network.

Differences between peer to peer and dedicated game server hosting. 2022. Blogi artikkeli Servers-sivustolla. Viitattu 03.04.2024. https://www.servers.com/news/blog/differences-between-peer-to-peer-and-dedicated-game-server-hosting.

Dreher, T. 2020. History of Computer Art. Morrisville/North Carolina: Lulu Press.

Fusion 2 Introduction. N.d. Dokumentaatio Photon Engine-sivustolla. Viitattu 07.01.2024. https://doc.photonengine.com/fusion/current/fusion-intro.

Game Developer May 2012 Issue. 2012. Lehti GDCVault-sivustolla. Viitattu 07.01.2024. https://ubm-twvideo01.s3.amazonaws.com/o1/vault/GD_Mag_Archives/GDM_May_2012.pdf.

Game server hosting (Multiplay). N.d. Sivu Unity-sivustolla. Viitattu 08.01.2024. https://unity.com/products/game-server-hosting.

Getting started. N.d. Dokumentaatio Unity-sivustolla. Viitattu 21.01.2024. https://docs.unity3d.com/Packages/com.unity.netcode@1.0/manual/getting-started.html.

Kanade, J. 2023. What Is Peer-To-Peer? Meaning, Features, Pros, and Cons. Artikkeli Spiceworks-sivustolla. Viitattu 03.04.2024. https://www.spiceworks.com/tech/networking/articles/what-is-peer-to-peer/.

Lag Compensation. N.d. Dokumentaatio PhotonEngine-sivustolla. Viitattu 09.04.2024. https://doc.photonengine.com/fusion/current/manual/advanced/lag-compensation.

McFadden, A. 1994. The History of Netrek, through Jan 1 1994. Artikkeli Netrek-sivustolla. Viitattu 03.12.2023. https://www.netrek.org/about/history_overall.php.

Mulligan, J. & Patrovsky, B. 2003. Developing Online Games: An Insider's Guide. Indianapolis: New Riders Pub. Viitattu 02.12.2023. https://www.researchgate.net/publication/234800904_Developing_Online_Games_An_Insider's_Guide, Researchgate.

Multiplayer networking challenges. 2023. Blogi artikkeli Argentics-sivustolla. Viitattu 05.04.2024. https://www.argentics.io/multiplayer-networking-challenges.

Neumann, C. 2007. Challenges in Peer-to-Peer Gaming. Artikkeli Computer Communication Review-sivustolla. Viitattu 05.04.2024. http://ccr.sigcomm.org/online/files/p2p_gaming.pdf.

Overview. N.d. Dokumentaatio Photon Engine-sivustolla. Viitattu 29.01.2024. https://doc.photonengine.com/quantum/current/quantum-100/overview.

Photon. N.d. Kotisivu Photon Engine-sivustolla. Viitattu 07.01.2024. https://www.photonengine.com/.

Prediction. 2023. Artikkeli ValveSoftware-sivustolla. Viitattu 09.04.2024. https://developer.valvesoftware.com/wiki/Prediction.

Pun & Bolt vs Fusion. N.d. Dokumentaatio Photon Engine-sivustolla. Viitattu 21.11.2023. https://doc.photonengine.com/fusion/current/getting-started/pun-bolt-vs-fusion.

Quantum 2 Intro. N.d. Dokumentaatio Photon Engine-sivustolla. Viitattu 07.01.2024. https://doc.photonengine.com/quantum/current/quantum-intro.

Realtime Intro. N.d. Dokumentaatio Photon Engine-sivustolla. Viitattu 07.01.2024. https://doc.photonengine.com/realtime/current/getting-started/realtime-intro.

Reconciliation in multiplayer games. 2023. Artikkeli Elympics-sivustolla. Viitattu 09.04.2024. https://docs.elympics.cc/concepts/reconciliation/index.html.

Source Multiplayer Networking. N.d. Artikkeli Valve-sivustolla. Viitattu 03.12.2023. https://developer.valvesoftware.com/wiki/Source_Multiplayer_Networking.

Spectre 1991 Video Game. N.d. Artikkeli Academic Accelerator-sivustolla. Viitattu 02.12.2023. https://academic-accelerator.com/encyclopedia/spectre-1991-video-game.

Toftedahl, M. 2019. Which are the most commonly used Game Engines?. Blogi artikkeli Game Developer-sivustolla. Viitattu 07.01.2024. https://www.gamedeveloper.com/production/which-are-the-most-commonly-used-game-engines-.

Tsaruk, V. 2023. How do multiplayer games work? From simple to complex. Artikkeli Gamestudio-sivustolla. Viitattu 05.04.2024. https://gamestudio.n-ix.com/how-do-multiplayer-games-work-from-simple-to-complex/.

Tuliper, A. 2015. Unity : Developing Your First Game with Unity and C#. Artikkeli Microsoft-sivustolla. Viitattu 08.01.2024. https://learn.microsoft.com/en-us/archive/msdn-magazine/2014/august/unity-developing-your-first-game-with-unity-and-csharp#architecture-and-compilation.

Ultima's Population Reaches 100,000. 2012. Artikkeli Ign-sivustolla. Viitattu 03.12.2023. https://www.ign.com/articles/1998/12/16/ultimas-population-reaches-100000.

Unity Technologies Acquires Game Hosting Division of Multiplay from GAME Digital, PLC. 2017. Uutinen Unity-sivustolla. Viitattu 09.04.2024. https://unity.com/our-company/newsroom/unity-technologies-acquires-game-hosting-division-multiplay-game-digital-plc.

What is a gaming server?. 2023. Artikkeli Servers-sivustolla. Viitattu 03.12.2023. https://www.servers.com/news/blog/what-is-a-game-server.

Wijman, T. 2023. New free report: Explore the global games market in 2023. Blogi artikkeli Newzoo-sivustolla. Viitattu 21.11.2023. https://newzoo.com/resources/blog/explore-the-global-games-market-in-2023.