Mikael Kuokkanen

Monialustainen kontitettu palvelinratkaisu 3D-ympäristössä sijaitsevan reaaliaikaisen datan esittämiseen

Metropolia Ammattikorkeakoulu

Insinööri (AMK)

Tieto- ja viestintätekniikka

Insinöörityö

16.5.2021

Tiivistelmä

Tekijä:	Mikael Kuokkanen
Otsikko:	Monialustainen kontitettu palvelinratkaisu 3D-ympäristössä sijaitsevan reaaliaikaisen datan esittämiseen
Sivumäärä:	56 sivua
Aika:	16.5.2021
Tutkinto:	Insinööri (AMK)
Tutkinto-ohjelma:	Tieto- ja viestintätekniikka
Ammatillinen pääaine:	Mobile Solutions
Ohjaajat:	Site Project Lead Mikael Lindblad Tutkijaopettaja Hannu Markkanen

---

Insinöörityön tarkoituksena oli toteuttaa kehitteillä olevaan PWA-sovellukseen (Progressive Web Application) integroitavissa oleva palvelinratkaisu jo olemassa olevalle reaaliaikaista dataa havainnollistavalle digitaalisen kaksosen (digital twin) Unity-projektille sekä jatkokehittää kyseistä projektia. Toteutuksessa pyrittiin myös tarjoamaan moninpelimuotoinen alustariippumaton interaktiivinen kokemus, jossa Unity-projektissa pystyisi reaaliaikaisen datan tarkastelun lisäksi sekä liikkumaan että olemaan jollain tapaa vuorovaikutuksessa muiden pelaajien kanssa.

Moninpelimuoto toteutettiin Unityn Mirror-kirjastoa käyttäen, ja eri versioille rakennettiin WebGL-alustalle aihiot, jotka pakattiin Docker-konttialustaa käyttäen Microsoft Azure -julkaisuympäristössä suoritettaviksi konteiksi. Docker-image-tiedostojen rakennus- ja julkaisuprosessi toteutettiin automatisoidusti GitHub Actions -palvelussa.

Projektissa käytetty 3D-malli todettiin kaiken optimoinnin jälkeen liian raskaaksi suorittaa selaimessa, joten interaktiivisten WebGL-versioiden sijaan päätettiin, että PWA-sovellukseen integroitava osuus tulisi olemaan vain reaaliaikainen suoratoisto projektiin tehdystä automatisoidusta ylilennosta, jossa reaaliaikaista dataa havainnollistetaan. Tämän lisäksi WebGL-versiot muunnettiin natiivimuotoisiksi ladattavissa oleviksi erillisiksi interaktiivisiksi sovelluksiksi.

Unity-projektin suoratoisto mahdollistettiin sulauttamalla siihen FFmpeg-multimediatyökalu, jota käyttäen lähetettiin projektissa renderöityä kuvaa RTMP-protokollalla sitä vastaanottavalle NGINX-palvelimelle. Palvelin konfiguroitiin julkaisemaan HLS-soittolistaa (HTTP Live Streaming) React-kirjastolla tehdyn asiakasohjelman käyttöön, josta suoraa lähetystä pystyi toistamaan verkkoselaimessa.

Insinöörityössä opituista käytännöistä saatiin selville suoratoiston kokonaisvaltaisen palvelun toteuttamisen monimutkaisuus sekä suurten 3D-mallien renderöimisen haasteet. Projektissa käytetty konttipohjainen ketterä kehitysmalli oli sekä arvokas oppi tekijän saaman taitotiedon että projektin jatkokehitysmahdollisuuksien kannalta. Tuloksena syntyi toimiva uudelleenkäytettävissä oleva ratkaisu.

Avainsanat: digitaalinen kaksonen, Unity, WebGL, Docker, FFmpeg, NGINX, RTMP, HLS, React, DevOps, suoratoisto

Abstract

Author:	Mikael Kuokkanen
Title:	A containerized multiplatform server solution for presenting real-time data in a 3D environment
Number of Pages:	56 pages
Date:	16 May 2021
Degree:	Bachelor of Engineering
Degree Programme:	Information and Communications Technology
Professional Major:	Mobile Solutions
Supervisors:	Mikael Lindblad, Site Project Lead Hannu Markkanen, Researching Lecturer

---

The objective of this thesis was both to produce a server solution and continue developing an existing real-time data visualizing digital twin Unity project that would be capable of being integrated into a PWA (Progressive Web Application) currently under development. The implementation attempted to offer an interactive multiplatform multiplayer experience, in which the player could move and have social interactions with other players in addition to visualizing the real-time data in the Unity project.

The multiplayer implementation was carried out using Unity's Mirror library. Every version was built for WebGL and containerized with Docker to be used in a production environment in Microsoft Azure. The build and deployment process of the Docker images was automated in GitHub Actions.

After numerous optimization attempts, the 3D model used in the Unity project was determined to be too heavy to run in a web browser, so it was decided that the part that would be integrated into the PWA, would be a live stream of an automated real-time data visualizing overflight built into the Unity project instead of the interactive WebGL builds. In addition, the WebGL builds were converted into separately downloadable interactive native applications.

The live streaming implementation was made possible by integrating an open-source multimedia tool, FFmpeg, into the Unity project. FFmpeg was used to send an RTMP (Real-Time Messaging Protocol) stream of the project's rendered image to an NGINX server that was configured to receive the stream and publish it as an HLS (HTTP Live Streaming) playlist. The HLS playlist was accessible from a media player built into a web client that was developed for this purpose using React.

This thesis provided the author with an understanding of the complexity of a complete streaming service implementation, and knowledge of the challenges that may arise when rendering large 3D models. The container-oriented agile development model used throughout the project was valuable both for the author's know-how and for the future development of the project. The result was a working reusable solution.

Keywords: digital twin, Unity, WebGL, Docker, FFmpeg, NGINX, RTMP, HLS, React, DevOps, live streaming

Sisällys

Lyhenteet

LTS: Long-term-support. Yleensä avoimen lähdekoodin ohjelmaversioiden yhteydessä käytetty vakaata versiota kuvaava lyhenne.

SSH: Secure Shell Protocol. Etäyhteyden mahdollistava salatun tietoliikenteen protokolla.

SCP: Secure Copy Protocol. SSH-protokollaan perustuva salatun tiedonsiirron mahdollistava protokolla.

UDP: User Datagram Protocol. Kuljetuskerroksen yhteydetön tietoliikenneprotokolla.

XR: Extended Reality. Yhteinen nimitys nykyisille (esim. VR ja AR) ja tuleville virtuaaliteknologioille.

SDK: Software Development Kit. Kokoelma työkaluja yleensä jonkin tietyn laitteen sovelluskehityksen helpottamiseksi.

RTMP: Real-Time Messaging Protocol. Multimedian verkon kautta lähettämiseen kehitetty tietoliikenneprotokolla.

HLS: HTTP Live Streaming. HTTP-pohjainen Applen kehittämä multimedian suoratoistoon sopiva tietoliikenneprotokolla.

1 Johdanto

Insinöörityö on osa Nokia Oyj:n Espoon-kampuksen työntekijöille kesällä 2021 julkaistavaa MyCampus-sovellusprojektia, jota Metropolia Ammattikorkeakoulun opiskelijat ovat olleet mukana kehittämässä.

Insinöörityön tärkeimpänä tavoitteena oli saada tuotettua MyCampus-sovellukseen integroitavissa oleva palvelinratkaisu, jossa kampuksen fyysisessä ravintolassa sijaitsevat ravintolaan asennettujen lämpökameroiden havaitsemat ihmiset simuloidaan reaaliaikaisesti ravintolan 3D-mallin pohjalta toteutettuun digitaaliseen ravintolaan animoiduiksi 3D-malleiksi. Projektin pohjana käytettiin Ilari Salosen vuonna 2020 tekemää insinöörityötä (Salonen 2020).

Insinöörityön toissijaisena tavoitteena oli tarjota MyCampus-sovelluksen käyttäjille interaktiivinen sosiaalinen kokemus toteuttamalla siinä käytetystä pohjaprojektista alustariippumaton moninpeliversio. Tuettaviksi alustoiksi valittiin web-mobiili- ja VR-alustat.

Raportissa perehdytään ensin työssä käytettyihin teknologioihin ja niiden yhteydessä käytettyihin kirjastoihin (luku 2). Tutkimustyöstä saadun tiedon perusteella syvennytään seuraavaksi työssä runsaasti käytettyyn konttiteknologiaan ja sen taustaan (luvut 3–4), minkä jälkeen selostetaan työnkulkua kronologisessa järjestyksessä.

Työn vaiheita selostaessa käytetään ajoittain taustatutkimuksessa selitettyä terminologiaa, eikä termejä tekstin yhtenäisyyden säilyttämisen vuoksi selitetä uudelleen, joten niistä voi ottaa uudelleen selvää palaamalla luvuissa 2–4 käsiteltyyn teoriaosuuteen. Työn vaiheista kertovien lukujen (luvut 5–6) yhteydessä arvioidaan jatkuvasti tehdyn työn tuloksia.

2 Insinöörityössä käytetyt teknologiat

Tässä luvussa esitellään insinöörityössä käytetyt teknologiat sekä niiden yhteydessä mahdollisesti käytetyt kirjastot käsitteellisellä tasolla.

2.1 Unity-pelimoottori

Unity (myös Unity3D) on pelimoottori, joka tarjoaa lukuisia korkean performanssin tarjoavia työkaluja sisältävän ympäristön pelien ja lukuisten 3D-sisältöä käsittelevien tai tuottavien palveluiden kehittämiseen ja julkaisuun monelle eri alustalle (Wiebe 2013).

Unity sisältää Unity Asset Store -kirjaston, josta voi ladata lukuisia muiden tekemiä 3D-malleja ja esimerkkiskriptejä, joista osa on ilmaisia (Wiebe 2013). Insinöörityössä käytettiin Asset Storesta löytyvää moninpelitoteutuksen mahdollistavaa ilmaista Mirror-kirjastoa. Mirror on helppokäyttöinen korkean tason rajapinta Unityn tietoverkko-operaatioille, ja siinä moninpelinä toteutettu asiakasohjelma ja palvelin ovat sekä saman projektin sisällä että jakavat saman koodin (Mirror Documentation).

Asset Storen lisäksi Unity tarjoaa oman Package Manager -paketinhallintajärjestelmän, josta voi ladata Unityn virallisia laajennuksia (Unity Manual). Package Managerista löytyy muun muassa projektiin asennettu XR Interaction Toolkit, joka on Unityn kehittämä ohjelmistokehys monialustaiselle XR-kehitykselle.

2.2 React-kirjasto

React on Facebookin kehittämä komponenttipohjainen JavaScript-kirjasto web-käyttöliittymien kehittämiseen. Reactissa käytetään JSX-ohjelmointikieltä, joka on JavaScript-ohjelmointikielen ja HTML-merkintäkielen yhdistämisen mahdollistava JavaScript-syntaksilaajennus. (React Documentation.)

Insinöörityössä käytettiin Reactia MyCampus-sovellukseen integroitavan käyttöliittymän luomiseen. Luodussa React-käyttöliittymässä käytettiin npm-paketinhallintatyökalun kautta ladattavaa react-player-mediasoitin-komponenttia, joka muun muassa mahdollistaa reaaliaikaiseen suoratoistoon käytettyjen videoformaattien web-selaimessa toistamisen.

2.3 FFmpeg-multimediatyökalu

FFmpeg on avoimen lähdekoodin komentoriviltä käytettävä multimediatyökalu, joka mahdollistaa lähes kaiken ihmisten ja koneiden luoman multimedian käsittelemiseen ja editoimiseen käytettävät toimenpiteet, riippumatta siitä, kuinka vanha formaatti on kyseessä. FFmpegin käyttäminen on mahdollista monella eri käyttöjärjestelmällä. (FFmpeg Documentation.)

Insinöörityössä käytettiin FFmpegia työssä käytetyn Unity-pohjaprojektin sisällä muun muassa projektissa renderöidyn kuvan sekä sopivaan formaattiin muuntamiseen että kuvan reaaliaikaiseen suoralähetykseen.

2.4 Node.js-ajoympäristö

Node.js on skaalautuvien verkkosovelluksien kehittämiseen suunniteltu asynkroninen tapahtumapohjainen JavaScript-ajoympäristö (About Node.js).

Node.js-projektiin voidaan ladata npm-paketinhallintatyökalun kautta lukemattomia ohjelmistokehyksiä erilaisille palvelintoteutuksille. Yksi näistä ohjelmistokehyksistä on insinöörityössä käytetty avoimen lähdekoodin Express.js-ohjelmistokehys, joka mahdollistaa erittäin yksinkertaisen web-palvelin-kehitysprosessin.

2.5 NGINX-palvelin

NGINX on ilmainen avoimen lähdekoodin sekä HTTP-palvelimena että käänteisenä välityspalvelimena toimiva web-palvelin, joka on tunnettu muun muassa sen korkeasta performanssista ja vakaudesta. NGINX on Node.js-ajoympäristön

tavoin asynkroninen, vähäisen muistinkäytön arkkitehtuurin omaava palvelin. Se mahdollistaa tuhansien samanaikaisten HTTP-pyyntöjen käsittelemisen. (NGINX Wiki.)

Insinöörityössä luotiin NGINX- käänteinen välityspalvelin moniin eri ympäristöihin ja käytettiin NGNIX-RTMP-moduulista tehdyn tutoriaalin (Enabling Video Streaming for Remote Learning with NGINX and NGINX Plus.) pohjaa FFmpegilla lähetetyn kuvan (luku 2.3) sekä vastaanottamiseen että suoratoistamiseen palvelimelta.

3 Virtualisointi

Jotta insinöörityössä käytettyä Docker-konttiteknologiaa olisi helpompi ymmärtää, täytyy ensin tietää, mihin se perustuu. Tässä luvussa tutustutaan kontittamisen peruskäsitteeseen, virtualisointiin: miten se toimii ja mitä ongelmia se ratkaisee.

Ennen virtualisoinnin aikaa, jotta pystyttiin julkaisemaan ja ylläpitämään sovelluksia luotettavalla ja tietoturvallisella tavalla, piti jokaiselle eri sovellukselle ostaa omat palvelimet omilla käyttöjärjestelmillään, joilla jokainen sovellus käytti vain pienen osan palvelimen resurssien kapasiteetista. Tämä ei ollut taloudellisesti kannattavaa, ja kapasiteettia meni hukkaan aina sitä enemmän, mitä enemmän uusia sovelluksia piti julkaista. Tämän ongelman ratkaisemiseksi ruvettiin kehittämään virtualisointia. (Virtualization 2019; What is virtualization.)

Virtualisointi on teknologia, joka mahdollistaa yhden tietokoneen fyysisten resurssien jakamisen moniksi eri toisistaan riippumattomiksi käyttöympäristöiksi, jotka tunnetaan virtuaalikoneina. Virtuaalikoneet pääsevät laitteen fyysisiin resursseihin käsiksi tähän kehitetyn ohjelmistokerroksen, hypervisorin avulla. Jokaisella virtuaalikoneella on oma käyttöjärjestelmänsä, ja vaikka ne käyttävätkin samoja fyysisiä resursseja, ne käyttäytyvät kuin oikeat yksilölliset tietokoneet. Hyvä analogia virtualisoinnille voidaan löytää teoksesta Virtualization Essen-

tials, jossa Portnoy (2012) vertaa tapaa, miten virtuaalikoneella sijaitseva sovellus kokee virtualisoinnin, meille vastaavaan virtuaalitodellisuuden tarjoamaan elämykseen, jossa sensorit ja visuaalinen projektio tarjoavat immersion, jossa kuvittelemme olevamme jossain muualla. Samalla tavalla tämä virtuaalikoneellakin suoritettava sovellus kuvittelee olevansa suoritettavana oikealla tietokoneella. (Portnoy 2012; Virtualization 2019; What is virtualization.)

Seuraavissa luvuissa käydään läpi virtualisoinnin teknisiä komponentteja ja siihen liittyviä konsepteja.

3.1 Hypervisor-ohjelmistokerros

Hypervisor on käyttöjärjestelmän päälle asennettava ohjelmistokerros, joka mahdollistaa yhden käyttöjärjestelmän ja sen resurssien jakamisen useammaksi virtuaalikoneeksi. IBM:n internetartikkelissa Virtualization (2019) hypervisoria kuvataan omanlaisekseen liikennepoliisiksi, jonka tehtävänä on ohjata kaikille vuoroaan odottaville virtuaalikoneille tarvittavat resurssit ja pitää huoli siitä, etteivät ne häiritse toisiaan. Hypervisoreita on kahdenlaista tyyppiä: Tyyppi 1 ja Tyyppi 2. (Virtualization 2019.)

Tyypin 1 hypervisorilla on suora yhteys jonkin tietokoneen fyysisiin komponentteihin, kuten sen muistiin, tallennustilaan ja prosessoriin. Tämäntyyppisiä hypervisoreita kutsutaankin usein bare-metal-hypervisoreiksi. Tämän suoran yhteyden vuoksi tyypin 1 hypervisor on hyvin tietoturvallinen, koska sen ja siihen yhteydessä olevan prosessorin välissä ei ole mitään, mihin voisi mahdollisesti hyökätä. Huono puoli tyypin 1 hypervisoreissa on se, että ne vaativat yleensä erillisen hallintalaitteen laitteistokomponenttien hallitsemiseen ja virtuaalikoneiden luomiseen. (Virtualization 2019.)

Toisin kuin tyypin 1 hypervisor, joka on suoraan kytköksissä tietokoneen laitteistokomponentteihin, tyypin 2 hypervisor suorittaakin ohjelmistoa tietokoneen käyttöjärjestelmässä. Tällaiset hypervisorit on kehitetty enemmän yksilöllisiä PC-käyttäjiä varten kuin julkaistujen sovellusten suorittamiseen palvelimella.

Tyypin 2 hypervisoreita käyttävät esimerkiksi tietoturva-ammattilaiset haittaohjelmien analysoimiseen. (Virtualization 2019.)

3.2 Virtuaalikoneet

Virtuaalikone on emulaatio fyysisestä tietokoneesta, joka toimii virtuaalisena tietokonejärjestelmänä. Virtuaalikoneeseen on virtualisoitu oma prosessori, tallennustila, muisti ja verkkosovittimensa sen tietokoneen resursseista, johon virtuaalikone on asennettu. Resurssien virtualisoinnin suorittaa luvussa 3.1 läpikäyty hypervisor. (Virtualization 2019; What is virtualization.)

Fyysistä tietokonetta, josta virtuaalikoneet luodaan, kutsutaan isännäksi (host), ja virtuaalikoneisiin viitataan nimellä vieras (guest). Isännällä voi olla monta eri vierasta, joista jokaisella voi olla eri virtuaalinen käyttöjärjestelmänsä, joiden käyttökokemus on lähes identtistä verrattuna fyysisellä tietokoneella sijaitsevaan vastaavan käyttöjärjestelmään. Jokainen vieras on eristyksissä muista vieraista, ja vieraita voidaan siirtää helposti muiden isäntien välillä (What is virtualization.)

4 Kontittaminen ja Docker-konttiteknologia

Luvussa 3.2 käytiin läpi virtuaalikoneita ja sitä, miten hypervisor-ohjelmistokerros luo niitä virtualisoimalla tietokoneen fyysisiä laitteistokomponentteja. Luvussa 4.1 tutustutaan samantapaiseen käsitteeseen, kontittamiseen (containerization), ja siihen, miten kontit eroavat virtuaalikoneista.

4.1 Kontittaminen

Kontti on oma eristetty käyttöympäristönsä, johon sovelluksen lähdekoodi ja siihen liittyvät kirjasto- ja ohjelmistokehysten riippuvaisuudet pakataan suoritusvalmiiksi. Kontit pitävät sisällään jopa oman käyttöjärjestelmänsä. Tämä mahdollistaa sen, että kontin voi viedä mihin vain sille yhteensopivaan uuteen ympäris-

töön, jossa konttiin pakatun sovelluksen voi suorittaa asentamatta mitään ylimääräistä. Kontit muistuttavat toimintamalliltaan hyvin paljon virtuaalikoneita, mutta niissä on pieniä eroja. Kontit eivät esimerkiksi tarvitse hypervisoria toimiakseen, vaan ne käyttävät hyväkseen käyttöjärjestelmätason virtualisointia, toiselta nimeltään kontittamista. (Virtualization 2019.)

Fyysisten laitteistokomponenttien virtualisoinnin sijaan kontittamisessa virtualisoidaan pelkästään fyysisen tietokoneen käyttöjärjestelmä. Tästä syystä kontit ovat paljon nopeampia kuin virtuaalikoneet, jotka käytännössä kopioivat koko tietokonejärjestelmän virtuaaliseen muotoon. Koska kontit pakkaavat sisälleen kaiken, mitä ne tarvitsevat siellä sijaitsevan sovelluksen suorittamiseksi, niitä on myös paljon yksinkertaisempaa siirtää uuteen ympäristöön. (Virtualization 2019.)

Kontittaminen alkaa olla lähes kaikkialla modernissa sovelluskehityksessä käytössä, eikä se näillä näkymin ole katoamassa mihinkään. Luvussa 4.2 käydään läpi konttiteknologiaa insinöörityössäkin paljon käytetyssä Docker-ympäristössä.

4.2 Docker-konttiteknologia

Docker on Go-ohjelmointikielellä kirjoitettu avoimen lähdekoodin konttialusta, joka julkaistiin vuonna 2013 (Mouat 2015; Get Started with Docker).

Docker-alustan toiminta perustuu käyttöjärjestelmällä taustalla suoritettavaan daemon-palveluprosessiin, dockerdiin ja komentoriviltä käytettävään Docker-asiakasohjelmaan. Dockerdin tehtävänä on konttien luomisen, valvomisen ja suorittamisen lisäksi Docker image -tiedostojen luominen ja tallentaminen. Ohjeiden antaminen dockerdille Docker-asiakasohjelmalta tapahtuu HTTP-protokollalla Dockerin tarjoamien rajapintojen kautta. (Mouat 2015; Get Started with Docker.)

Docker image -tiedosto

Image on Docker-kontissa suoritettava tiedosto, jossa on ohjeet kontin sisällön luomiseen. Image pitää sisällään koko sovelluksen ja siihen liittyvät riippuvuudet sekä viitteen sen suorittamiseksi vaadittavaan käyttöjärjestelmään. Käyttöjärjestelmä itsessään on myös julkisesta Docker-rekisteristä ladattava image, ja imagejen taustalla onkin usein jokin toinen image. (Mouat 2015; Get Started with Docker.)

Dockerfile-konfiguraatiotiedosto

Dockerfile on YAML-merkintäkielellä kirjoitettava konfiguraatiotiedosto, joka pitää sisällään ohjeet imagen luomisprosessiin. Ohjeet kirjoitetaan allekkaisiin riveihin, sellaiseen muotoon, jossa jokainen rivi voitaisiin suorittaa myös suoraan Docker-asiakasohjelman komentoriviltä. Dockerfile suoritetaan komentoriviltä docker-build-komennolla, joka käy tiedoston jokaisen rivin läpi ja tämän perusteella tuottaa uuden Docker imagen. Jokainen ohjerivi luo oman kerroksensa Docker imageen, ja jos imagen uudelleenrakentaa Dockerfilen muuttumisen jälkeen, rakennetaan pelkät muuttuneet kerrokset uudelleen. (Mouat 2015; Get Started with Docker.)

Docker-rekisterit

Docker-rekisterit ovat imagejen jakamiseen ja tallentamiseen tarkoitettuja palveluita. Docker käyttää tähän tarkoitukseen oletuksena omaa pilvipalveluaan, Docker Hubia (Get Started with Docker).

Docker Hub on julkinen rekisteri, jota kuka tahansa voi käyttää ilmaisen Docker-tilin luotuaan. Docker Hubissa on suuri määrä julkisia imageja, joita jokainen voi ottaa käyttöön omaan ympäristöönsä, sen sijaan että kehittäisi jotain jo olemassa olevaa ratkaisua itse alusta loppuun. Docker Hubiin on myös mahdollista luoda yksityisiä rekisterejä, jotka näkyvät oletuksena vain ne luoneelle käyttäjälle. Jokaiselle käyttäjälle on saatavilla yksi ilmainen yksityinen rekisteri, mutta tarpeen tullen voi liittyä Docker Pro -jäseneksi viiden dollarin kuukausimaksua vastaan saadakseen rajattoman määrän niitä käyttöönsä. (Mouat 2015; Get Started with Docker; Docker Subscriptions and Billing FAQs.)

Luvuissa 2–4 käsitellystä teoriaosuudesta saatiin hyvä pohja insinöörityössä käytettyjen teknologioiden ymmärtämiselle. Luvuissa 5–6 sovelletaan teoriaa käytännössä.

5 Aihion luominen

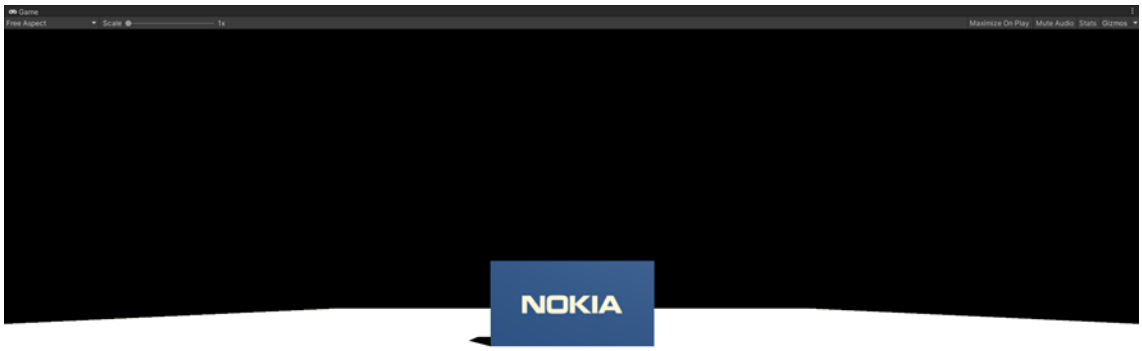
Työ aloitettiin tekemällä omaan projektiin aihio, jonka tarkoituksena oli pitää sisällään kaikki oleelliset toiminnallisuudet, jotka tälle projektille todettiin tarpeellisiksi. Tällä tavalla voitiin kokeilla ja kehittää eri ratkaisuja, joita käytettäisiin tuotantoprojektissa. Myös mahdolliset huonot menetelmät olivat nähtävillä aikaisessa vaiheessa. Aiempi, vuonna 2020 Ilari Salosen insinöörityössä tehty ravintolaprojekti (Salonen 2020) toteutettiin 2019.2.11f1-Unity-versiolla, joten tämäkin projekti oli järkevää toteuttaa 2019-alkuisella versiolla. Lopulta päädyttiin valitsemaan 2019.4.18f1-versio, joka on yksi Unityn LTS-versioista (long-term-support).

Aihioprojektin päätavoitteena oli saada kehitettyä toteutus, joka sisältäisi vähintään pelaajan liikkumisen virtuaalisessa ravintolassa ja jokaisen alustan (VR, mobiili ja desktop) asiakasohjelmat ja pääpalvelimen yhtenä kokonaisuutena käynnistettävässä Docker-ympäristössä.

5.1 Yksinkertainen maisema ja moninpelitoteutus

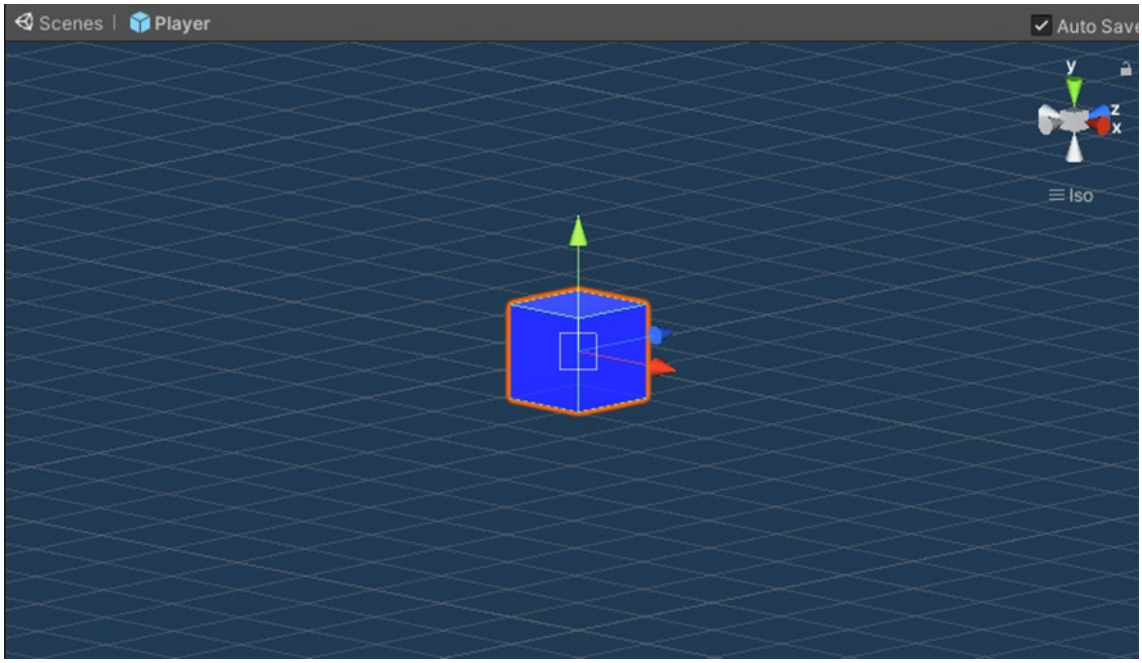
Aihioprojekti aloitettiin suunnittelemalla ja lataamalla tarvittavat Unity-riippuvuudet sekä moninpelitoteutuksen että web-asiakasohjelmalla suorittamisen mahdollistamiseksi. Moninpelitoteutukseen valittiin Mirror-kirjasto ja versioiden rakentamisalustaksi valittiin WebGL.

Aluksi luotiin yksinkertainen maisema, jota käytettiin pohjana jokaiselle eri versiolle. Maisemaan tehtiin mustalle taustalle valkoisella tasolla sijaitseva sininen laatikko, joka oli varustettu Nokian logolla (kuva 1). Tällä tavoin luotiin maisemaan pientä tunnelmaa, jotta sen esittäminen viikoittaisissa kokouksissa olisi katsojan näkökulmasta mielekkäämpää.



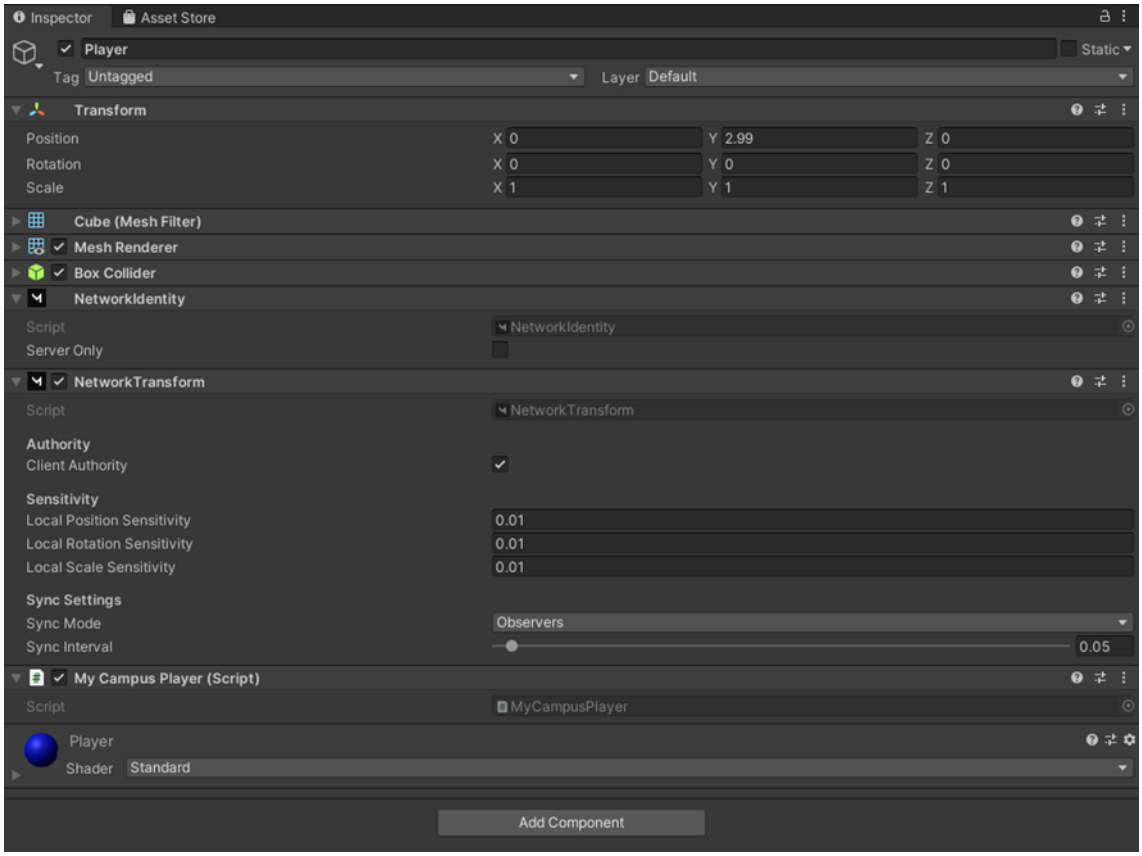
Kuva 1. Aihion yksinkertainen maisema.

Maisemaan luotiin pelaajaobjekti, jonka avulla pystyttiin esittämään haettua interaktiivista kokemusta jo varhaisessa vaiheessa. Pelaajaobjektista tehtiin sininen kuutio, jonka avulla pystyttiin havaitsemaan monta eri pelaajaa samassa maisemassa. (Kuva 2.)



Kuva 2. Aihioprojektin pelaajaobjekti.

Jotta maisemaan saatiin todellista interaktiota, ei riittänyt, että siihen lisättiin pelkkä kuutio edustamaan pelaajaa, vaan pelaajaobjektille täytyi antaa muutamat skriptit (kuva 3).



Kuva 3. Aihion pelaajaobjektin skriptit.

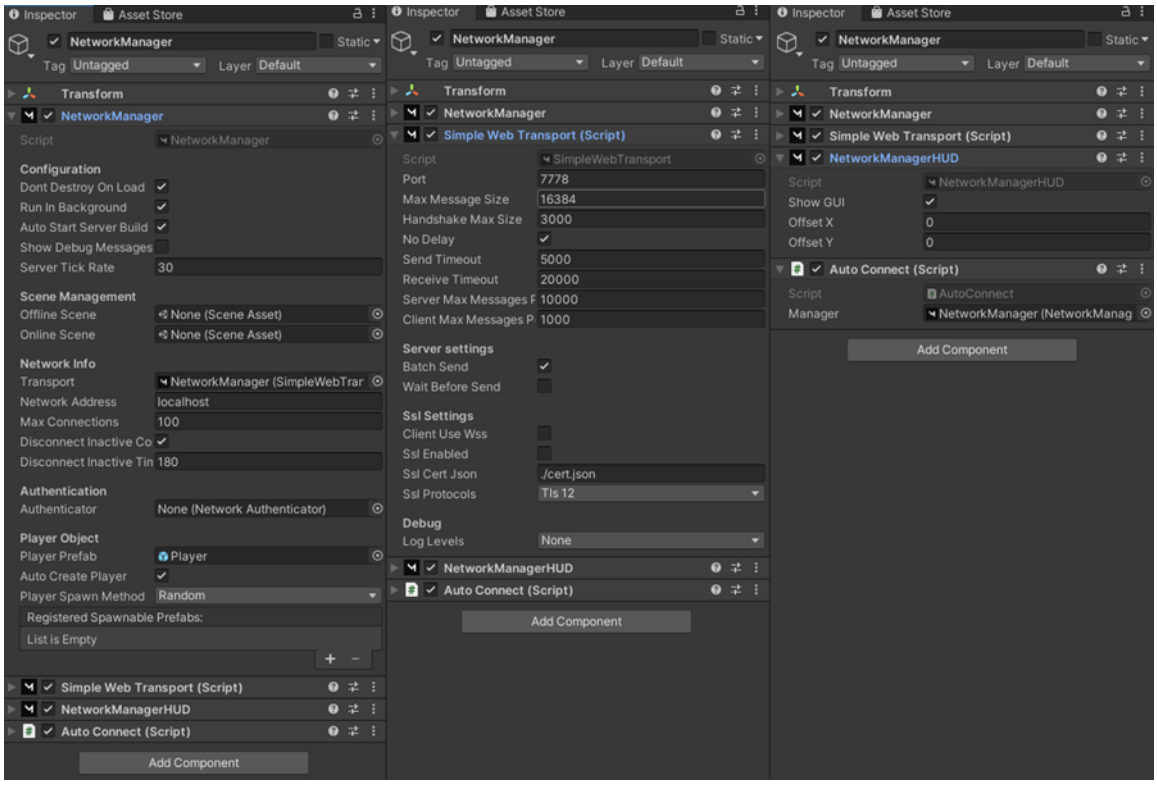
Mirror-kirjaston NetworkIdentity-skriptin avulla saadaan määritettyä kyseinen peliobjekti sellaiseksi peliobjektiksi, mikä mahdollisesti renderöidään jokaisen pelaajan nähtäväksi. NetworkTransform-skripti huolehtii pelaajaobjektin synkronoinnista muiden pelaajien ruudulla. (Kuva 3.)

Pelaajan liikkumista varten luotiin MyCampusPlayer-skripti, joka kuuntelee pelaajan syötettä ja liikkuu sen perusteella vakioon asetetulla nopeudella oikeaan suuntaan. Jotta pelaajat eivät pystyisi hallitsemaan muiden pelaajaobjektien sijaintia, käytettiin liikkumisfunktiolle ehtona Mirror-kirjaston isLocalPlayer-muuttujaa. Muuttuja isLocalPlayer varmistaa, että skripti on osa sen käyttäjän asiakasohjelmaa, jolta syöte tuli. (Esimerkkikoodi 1.)

```
using Mirror;
using UnityEngine;

public class MyCampusPlayer : NetworkBehaviour {
    private const float movSpeed = 10;

    void Update() {
        HandleMovementInput();
    }

// Listen to horizontal and vertical axes' input and move the player
object accordinly.

    private void HandleMovementInput() {
        if (isLocalPlayer) {
            float moveHorizontal = Input.GetAxis("Horizontal");
            float moveVertical = Input.GetAxis("Vertical");
            float modifier = Time.deltaTime * movSpeed;
            Vector3 movement = new Vector3(
              moveHorizontal * modifier, moveVertical * modifier, 0);
            transform.position = (transform.position + movement);
        }
    }
}
```

Esimerkkikoodi 1. Pelaajan liikkumista ohjaava MyCampusPlayer-skripti.

Mirror-kirjaston käyttämisen helpottamiseksi on olemassa NetworkManager-skripti, jonka käyttäminen jo itsessään mahdollistaa pelin moninpelitoteutuksen. NetworkManager-skriptille luotiin oma peliobjekti, johon se asetettiin. Peliobjektiin lisättiin tämän lisäksi tarvittavat skriptit WebGL-yhteensopivuudelle. (Kuva 4.)



Kuva 4. NetworkManager-peliobjektiin asetetut moninpelitoteutuksen mahdollistavat skriptit.

Mirror-kirjaston oletusskriptien lisäksi luotiin AutoConnect-skripti, joka yhdistää jokaisen pelaajan automaattisesti pääpalvelimelle. Skriptissä tarkistetaan ensin, onko kyseessä pääpalvelin (isBatchMode) vai asiakasohjelma. Pääpalvelintilanteessa laitetaan ohjelmaa suorittaessa palvelin automaattisesti käyntiin, ja asiakasohjelmatilanteessa yhdistetään palvelimeen automaattisesti. (Esimerkkikoodi 2.)

```
using UnityEngine;
using Mirror;

public class AutoConnect : NetworkBehaviour {
    [SerializeField]
    NetworkManager manager;

    public string HostAddress { get => manager.networkAddress; }

    private void Start() {
        // Automatically connect to the server when not a headless
build
        if (!Application.isBatchMode) {
            Connect();
        } else {
            manager.StartServer();
            Debug.Log("Starting server...");
            Debug.Log(HostAddress);
        }
    }

    private void Connect() {
        manager.networkAddress = "localhost";
        Debug.Log("Connecting to the server: " + HostAddress + "...");
        Debug.Log("Client platform: " + Application.platform);
        manager.StartClient();
    }
}
```

Esimerkkikoodi 2. NetworkManager-peliobjektiin liitettävä AutoConnect-skripti asiakasohjelman automaattiselle pääpalvelimelle yhdistämiselle.

Aiemmissa työnvaiheissa luodusta Unity-projektista rakennettiin kaiken kaikkiaan kolme eri alustoille optimoitua WebGL-versiota. Vaikka projektiin ei vielä tässä vaiheessa ollut tehty erillisiä käyttöliittymiä mobiilille ja VR:lle, niin niille koettiin hyväksi olla jo omat eristetyt pohjat valmiina. WebGL-versioiden lisäksi rakennettiin pääpalvelimena toimiva Linux-versio.

5.2 WebGL-versioiden Node.js-palvelimet ja kontittaminen

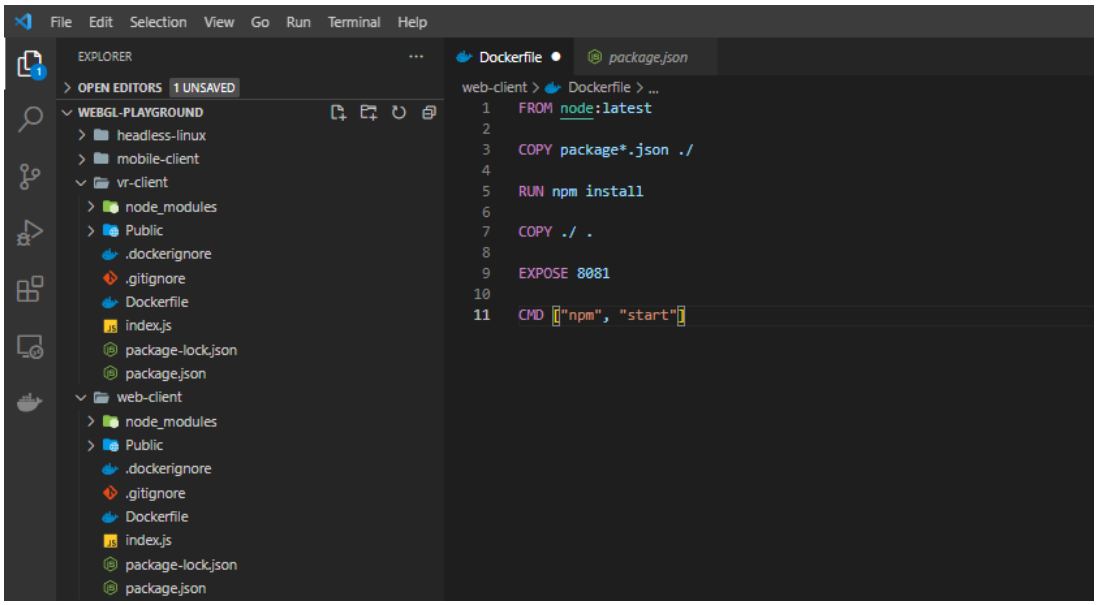
Jotta WebGL-versioita pystyttiin suorittamaan muualta kuin suoraan tiedostosta, täytyi niille luoda jonkinlainen palvelin. Tähän tarkoitukseen luotiin yksinkertainen Node.js-palvelin, jonka tehtävänä oli tarjota kyseiset WebGL-versioiden tiedostot sille konfiguroidulta portilta HTTP-protokollaa käyttäen (esimerkkikoodi 3).

```
const express = require('express');
const path = require("path");
const app = express();

// Web client
app.use("/web", express.static(path.resolve(__dirname, 'Public')));

app.get('/web', (req, res) => {
    res.sendFile(
            path.join(__dirname, 'Public', "Build", 'index.html'));
});

//Launch listening server on port 8081
app.listen(8081, () => {
    console.log('app listening on port 8081!');
});
```

Esimerkkikoodi 3. WebGL-versiota tarjoava Node.js-palvelin.

Node.js-palvelimelle ladattiin npm-paketinhallintajärjestelmän kautta Express-ohjelmistokehys, jota käyttämällä saatiin luotua web-palvelin mahdollisimman vähällä vaivalla. Palvelin monistettiin jokaiselle eri versiolle tarjoamaan niitä eri porteilta.

Aihioprojektista luodut WebGL-versiot ja niitä vastaavat Node.js-palvelimet sekä Linux-versio järjesteltiin omiin kansoihinsa, ja jokaisen kansion sisään luotiin Dockerfile-tiedosto, jossa kopioidaan ja asennetaan version riippuvuudet sekä paljastetaan version käyttämä portti (kuva 5).



Kuva 5. WebGL-versioiden tiedostorakenne havainnollistettuna.

Versioiden palvelemiseksi suoraan web-osoitteesta määrittämättä porttia luotiin sekä käänteisenä välityspalvelimena toimiva NGINX-konfiguraatiotiedosto (esimerkkikoodi 4) että käänteisen välityspalvelimen kontittamisen mahdollistava Dockerfile (esimerkkikoodi 5).

```
events {
    worker_connections 1024;
}

http {

    # Define HTTP server and redirect HTTP to HTTPS
    server {
        listen 80 default_server;
        return 301 https://$request_uri;
    }

    server {
        # Define SSL server
        listen 443 ssl default_server;

        ssl_certificate /etc/nginx/ssl/certificate.crt;
        ssl_certificate_key /etc/nginx/ssl/certificate.key;

        # Serve nginx default index.html from base endpoint
        location / {
            root /usr/share/nginx/html;
            index index.html index.html;
        }

        # Define reverse proxy for WebGL build optimized for Desktop
        location /web {
            proxy_pass http://web:8081;
        }

        # Define reverse proxy for WebGL build optimized for VR
        location /vr {
            proxy_pass http://vr:8082;
        }

        # Define reverse proxy for WebGL build optimized for mobile
        location /mobile {
            proxy_pass http://mobile:8083;
        }

        # Define reverse proxy for headless Unity server
        location /server {
            proxy_pass http://server:7778;
        }
    }
}
```

Esimerkkikoodi 4. Käänteisen välityspalvelimen ja WebGL-versioiden sekä tarjoamiseen että pääpalvelimelle yhdistämiseen mahdollistava NGINX-konfiguraatiotiedosto, joka myös ohjaa HTTP-liikenteen HTTPS-liikenteeksi.

```
FROM nginx

RUN mkdir /etc/nginx/ssl

RUN chmod 700 /etc/nginx/ssl

RUN openssl req -new -newkey rsa:4096 -days 3650 -nodes -x509 -batch -
keyout /etc/nginx/ssl/certificate.key -out /etc/nginx/ssl/certifi-
cate.crt

COPY nginx.conf /etc/nginx/nginx.conf
```

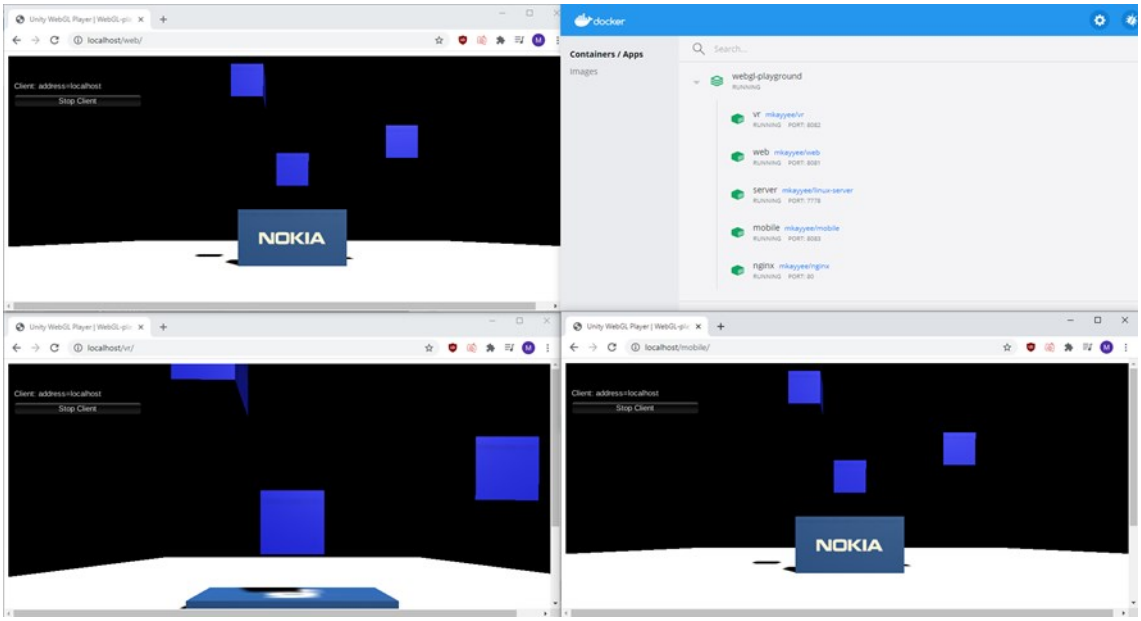
Esimerkkikoodi 5. NGINX- käänteisen välityspalvelimen Dockerfile-tiedosto, jossa luodaan itse allekirjoitettu SSL-sertifikaatti sekä kopioidaan NGINX-konfiguraatiotiedosto Docker-image-tiedoston sisään Dockerfilea suorittaessa.

Kaikki Dockerfile-tiedoston sisältämät komponentit konfiguroitiin yhtenäiseksi kokonaisuudeksi docker-compose-tiedoston avulla (esimerkkikoodi 6).

```
services:
  # Nginx reverse proxy from an nginx.conf file
  nginx:
    image: nginx
    container_name: nginx
    build:
      context: ./nginx
    depends_on:
      - web
      - mobile
      - vr
    volumes:
      - ./nginx/nginx.conf:/etc/nginx/nginx.conf
    ports:
      - 80:80
      - 443:443

  # A headless linux server running on ubuntu
  server:
    image: linux-server
    container_name: server
    build:
      context: ./headless-linux
    ports:
      - 7778:7778
    restart: on-failure

  # Unity WebGL web client
  web:
    image: web
    container_name: web
    build:
      context: ./web-client
    ports:
      - 8081:8081
    restart: on-failure

  # Unity WebGL VR client
  vr:
    image: vr
    container_name: vr
    build:
      context: ./vr-client
    ports:
      - 8082:8082
    restart: on-failure

  # Unity WebGL mobile client
  mobile:
    image: mobile
    container_name: mobile
    build:
      context: ./mobile-client
    ports:
      - 8083:8083
    restart: on-failure
```

Esimerkkikoodi 6. Docker-compose-tiedosto, joka rakentaa aihioprojektin eri komponentit yhdeksi kokonaisuudeksi.

Docker-compose-tiedoston konfiguroimisen jälkeen koko projektin pystyi suorittamaan docker-compose build- ja up-komennoilla, joiden tulos näkyy kuvassa 6.



Kuva 6. Lokaalissa verkossa käynnissä oleva aihioprojektin kokonaisuus.

Insinöörityössä oli tähän mennessä saatu modulaarinen, kaikki sovelluksen julkaisuun vaadittavat osat sisältävä, web-selaimessa suoritettava moninpelimuotoinen kokonaisuus, jossa projektiin suunnitellut käytännöt oli todettu toimiviksi lokaalissa ympäristössä. Seuraavaksi kokeiltiin sen toimivuutta julkaisuympäristössä.

5.3 Jatkuva toimitus Microsoft Azure -virtuaalikoneelle

MyCampus-sovelluksen kehitystiimillä oli julkaisuympäristönä käytössä Microsoft Azure -pilvipalvelu, jonne luotiin oma virtuaalikone aihioprojektille. Virtuaalikoneeksi valittiin Azure Marketplace -palvelusta Linux-käyttöjärjestelmäpohjainen CentOS 7.7 -virtuaalikone, johon oli Docker-alusta jo valmiiksi asennettuna. Koska projekti oli tässä vaiheessa vain konseptin testaamista varten, laitteis-

toksi riitti vain yksi virtuaalinen prosessori ja 3.5 GB muistia. Jotta Mirror-pääpalvelimelle voitiin yhdistää, piti virtuaalikoneelta avata palvelimen käyttämä 7778-portti.

Virtuaalikoneen luomisen jälkeen suoritettiin esimerkkikoodi 7:ssä esitetyt komennot. Komennoissa rakennetaan aiemmin esitetyssä (esimerkkikoodi 6) docker-compose-tiedostossa määritetyt docker-imaget ja julkaistaan ne yksityiseen Docker-rekisteriin. Tämän jälkeen kopioidaan compose-tiedoston pohjalta luotu julkaisuympäristöön sopiva compose-tiedosto virtuaalikoneelle käyttämällä SCP-protokollaa (Secure Copy Protocol) ja otetaan SSH-yhteys (Secure Shell Protocol) virtuaalikoneeseen. Virtuaalikoneelle asennetaan docker-compose ja sille annetaan suoritusoikeudet, minkä jälkeen virtuaalikoneen käyttäjä lisätään docker-ryhmään, jotta docker-komentojen suoritus olisi mahdollista ilman sudo-komentoa. Lopuksi kirjaudutaan sisään Docker-rekisteriin, josta Docker-imaget tuodaan virtuaalikoneelle suoritukseen.

```
$ docker-compose build

$ docker-compose push

$ scp compose-prod.yaml azureuser@xx.xx.xxx.xx/:/home/azureuser

$ ssh azureuser@xx.xx.xxx.xx

$ sudo curl -L "https://github.com/docker/compose/releases/down-
load/1.28.2/docker-compose-$(uname -s)-$(uname -m)" -o /usr/lo-
cal/bin/docker-compose

$ sudo chmod +x /usr/local/bin/docker-compose

$ sudo usermod -aG docker azureuser

$ docker login

$ docker-compose -f compose-prod.yaml up
```

Esimerkkikoodi 7. Docker-konttikokonaisuuden rakentaminen ja julkaiseminen Docker Hubiin ja sen vieminen Azuren virtuaalikoneelle.

Julkaisuympäristön virtuaalikoneen konfiguroimisen jälkeen varmistettiin, että kaikki komponentit toimivat samalla tavalla kuin lokaalistikin. Komponentit toimivat odotetulla tavalla, joten voitiin aloittaa jatkuvan integraation ja julkaisun suunnitteleminen virtuaalikoneelle.

Ennen aihioprojektin käytäntöjen integroimista tuotantoprojektiin kehitysprosessista päätettiin tehdä mahdollisimman automatisoitu. Projektille luotiin versionhallintaa varten oma GitHub-repositorio, ja jatkuvaa julkaisua ja integrointia varten otettiin käyttöön GitHub Actions -palvelu. Palveluun luotiin jokaisen version julkaisun yhteydessä suoritettava YML-tiedosto (esimerkkikoodi 8), johon konfiguroitiin suoritettavaksi käytännössä samat komennot, joita esimerkkikoodi 7:ssä käytettiin. Arkaluotoiset tiedot kuten virtuaalikoneen IP-osoite, virtuaalikoneen SSH-yhteyden salliva julkinen avain ja Docker-käyttäjätiedot tallennettiin GitHubin tarjoamiin salattuihin GitHub secrets -ympäristömuuttujiin.

```
name: Docker-Compose Build/Push

# -> Builds and pushes docker images to a private repository for each
component
# -> Connects via ssh to a virtual machine in Azure
# -> Stops the running containers on the VM and pulls the new images
# -> Sets all the containers running again in detach mode

on:
  push:
    branches: [ main ]
  pull_request:
    branches: [ main ]

  workflow_dispatch:

env:
  DOCKER_USER: ${{ secrets.DOCKER_USER }}
  DOCKER_PASSWORD: ${{ secrets.DOCKER_PASSWORD }}

jobs:
  build:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v2
      - name: docker login
        run: |
          docker login -u $DOCKER_USER -p $DOCKER_PASSWORD
      - name: build images
        run: |
          docker-compose build
      - name: push images
        run: |
          docker-compose push
      - name: scp docker-compose prod
        uses: appleboy/scp-action@master
        with:
          host: ${{ secrets.VM_IP }}
          username: ${{ secrets.VM_USER }}
          key: ${{ secrets.SSH_KEY }}
          source: docker-compose-prod.yaml
          target: /home/${{ secrets.VM_USER }}
      - name: SSH and docker-compose on remote
        uses: appleboy/ssh-action@master
        with:
          host: ${{ secrets.VM_IP }}
          username: ${{ secrets.VM_USER }}
          key: ${{ secrets.SSH_KEY }}
          port: 22
          script: |
            docker login -u ${{ secrets.DOCKER_USER }}
                         -p ${{ se-crets.DOCKER_PASSWORD }}
            docker-compose -f docker-compose-prod.yaml down
            && docker-compose -f docker-compose-prod.yaml pull
            && docker-compose -f docker-compose-prod.yaml up -d
```

Esimerkkikoodi 8. GitHub Actions -palvelussa suoritettava konfiguraatiotiedosto, joka rakentaa Docker-imaget ja vie ne SSH-yhteydellä Azuren virtuaalikoneelle, jossa ne suoritetaan docker-compose-komennoilla.

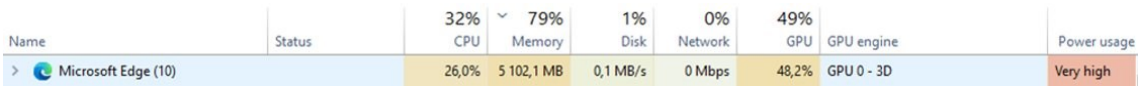
Aihioprojektista oli tässä vaiheessa saatu jatkuvan integraation ja julkaisun sisältävä pohja moninpelitoteutukselle ja alustava varmuus WebGL-versioiden toimivuudelle myös julkaisuympäristössä. Minkäänlaisia ongelmia ei tässä vaiheessa ollut ilmaantunut, joten seuraavaksi päätettiin integroida kehitetyt käytännöt tuotantosovellukseen.

6 Tuotantosovellusprojektiin siirtyminen

Vaikka WebGL-moninpeliversiot toimivat moitteettomasti aihioprojektissa, samojen käytäntöjen toteutus varsinaisessa tuotantoprojektissa ei ollut yhtä suoraviivainen, kuin sen alun perin oletettiin olevan.

6.1 Ongelmat ja maiseman optimointi

Aihioprojektin käytäntöjen tuominen tuotantoprojektiin ei integrointivaiheessa tuottanut minkäänlaisia ongelmia, mutta ravintolan 3D-mallin sisältämää tuotantoprojektia suorittaessa selaimessa tuli kriittinen ongelma vastaan: selain käytti yli 5 GB muistia (kuva 7). Rakennetut versiot olivat tämän lisäksi kooltaan noin 700 MB, mikä ylitti GitHubin asettaman 100 MB -rajoituksen yksittäisen tiedoston koolle, joten myöskään aiemmin toteutetun kontteja automaattisesti rakentavan ja julkaisevan CI/CD-putkiston käyttäminen ei ollut enää mahdollista.

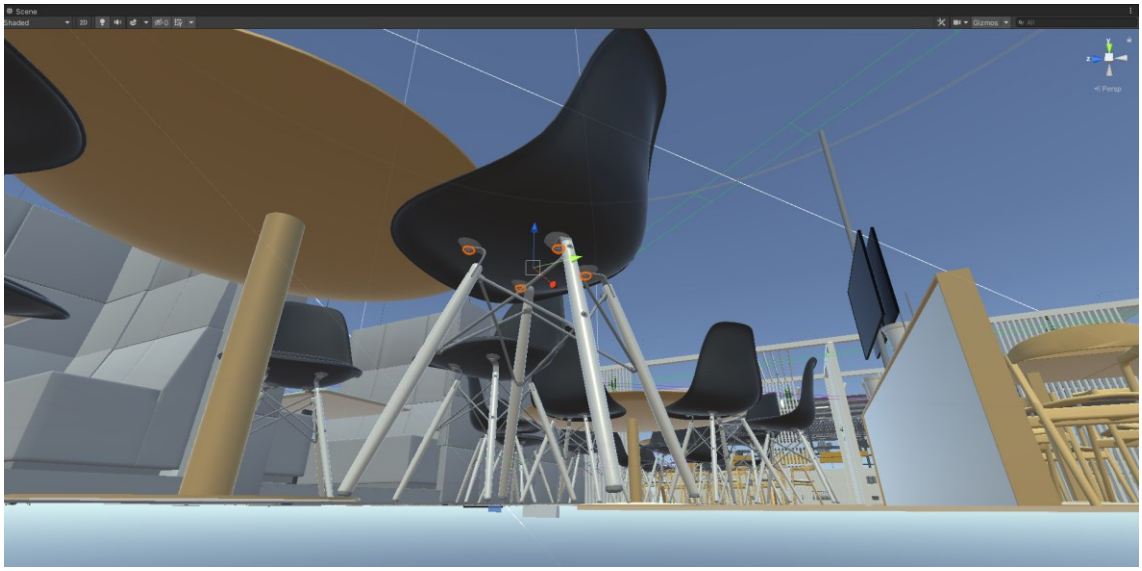


Kuva 7. Tuotantosovellusprojektin WebGL-version käyttämät resurssit kannettavalla tietokoneella, jossa on Intel i7-9750H-prosessori ja GeForce RTX 2060 -näytönohjain.

Liiallisten resurssien käytön vuoksi uudet WebGL-versiot eivät enää toimineet halutulla tavalla, vaan maisemassa liikkuminen oli jatkuvan pätkimisen vuoksi erittäin epämukavaa. Koska sulavan käyttökokemuksen saavuttaminen olisi asettanut tulevaisuuden käyttäjäryhmälle liian suuret laitteistovaatimukset, maisema oli välttämätöntä saada jollain tapaa optimoitua.

Ravintolasta tehty FBX-tiedostopäätteinen 3D-malli oli kooltaan noin 1,5 GB, ja se koostui noin 6000:sta eri objektista. Maiseman optimointi aloitettiin kokeilemalla poistaa sellaisia objekteja, jotka eivät tuottaneet niin suurta visuaalista merkitystä.

Mallista oli tehty erittäin yksityiskohtainen. Sen sisältämiin tuoleihin oli esimerkiksi suurimmalta osin jopa jokaisesta ruuvista tehty omat objektit, eivätkä ne välttämättä olleet edes millään tapaa näkyvillä. (Kuva 8.)



Kuva 8. Näkymä Unity-tuotantoprojektista, jossa on tuolin istuinosan alla olevat ruuvit valittuna.

Vaikka 3D-mallissa oli selvästi paljon mahdollista karsittavaa, sitä hankaloitti mallista koostuvien objektien määrä. Kaikki 3D-mallin noin 6 000 objektia olivat enimmäkseen samalla hierarkiatasolla, eikä niitä ollut nimetty mitenkään loogisella tavalla.

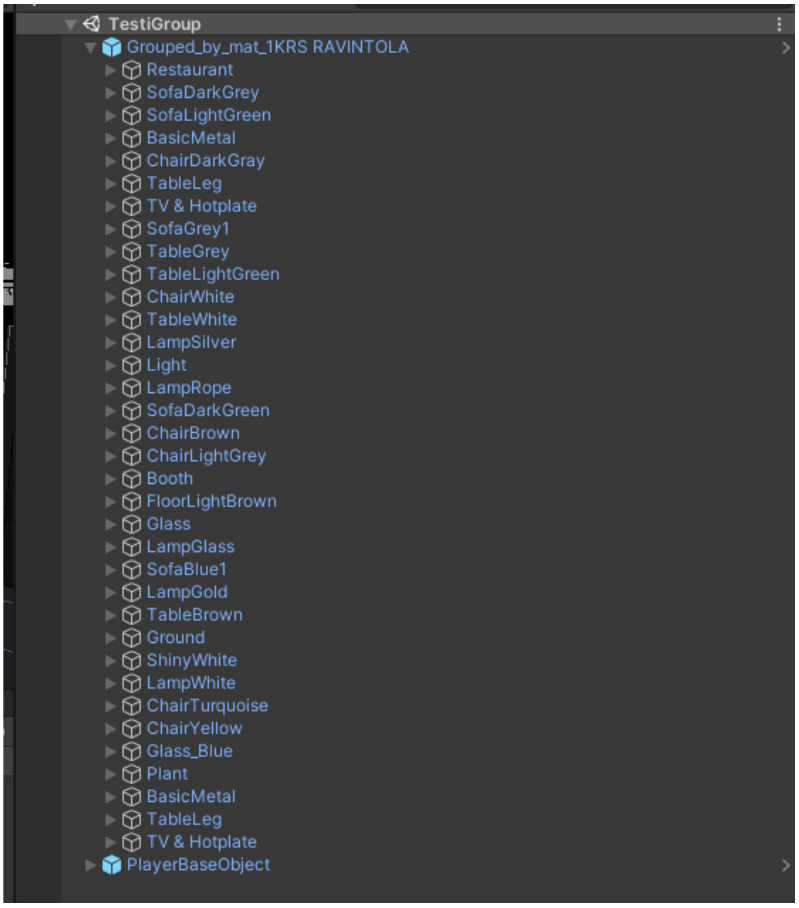
Objekteja koetettiin aluksi ryhmittää manuaalisesti, mutta tämä käytäntö ei ollut kovinkaan tehokas, joten sitä varten kehitettiin materiaalien perusteella kaikki objektit ryhmittävä Editor-skripti (esimerkkikoodi 9). Editor-skriptit ovat Unityssa skriptejä, joilla voi räätälöidä Unity-näkymää omilla skripteillä, esimerkiksi lisäämällä painikkeen, joka suorittaa jonkin halutun toimenpiteen.

```
// Groups all children of a selected
// game object based on their material

    [MenuItem("Grouping/Group children")]
    public static void GroupChildren() {
        // Currently selected GameObject in the scene
        GameObject parent = Selection.activeGameObject;

        Debug.Log("Beginning to group "
           + parent.transform.childCount + " children...");

        // All child material names in string format
        List<string> materialNames = new List<string>();

        // A list of root objects for different materials
        List<GameObject> rootObjects = new List<GameObject>();

        int childCount = parent.transform.childCount;

        // index that is incremented only
        // if a material is missing.
        int k = 0;

        for (int i = 0; i < childCount; i++) {
            // Every child that has a material is moved into
            // a new hierarchy, so always get the first index.
            Transform child = parent.transform.GetChild(k);

            // Get child transform's renderer
            Renderer renderer = child.GetComponent<Renderer>();

            // Some children do not have a renderer,
            // so we need a null check here
            if (renderer) {
                string mName = renderer.sharedMaterial.name;

                // Add child's material name to the list of
                // all material names if not already in it.
                if (!materialNames.Contains(mName)) {
                    materialNames.Add(mName);

                    // Create a root GameObject for every material.
                    GameObject root = new GameObject(mName);
                    rootObjects.Add(root);

                    // Assign the material root object to the
                    // iterated GameObject's hierarchy.
                    root.transform.parent = parent.transform;
                }

                // Move children to new parents
                // named after their material.
                foreach (GameObject r in rootObjects) {
                    if (r.name == mName) {
                        child.transform.parent = r.transform;
                        break;
                    }
                }
            } else {
                Debug.LogError("Transform "
```

```
                + child.name + " does not have a renderer.");
                  // Increment current index “k” when a child does
                  // not move to a new hierarchy.
                  k++;
              }
          }
      }
```

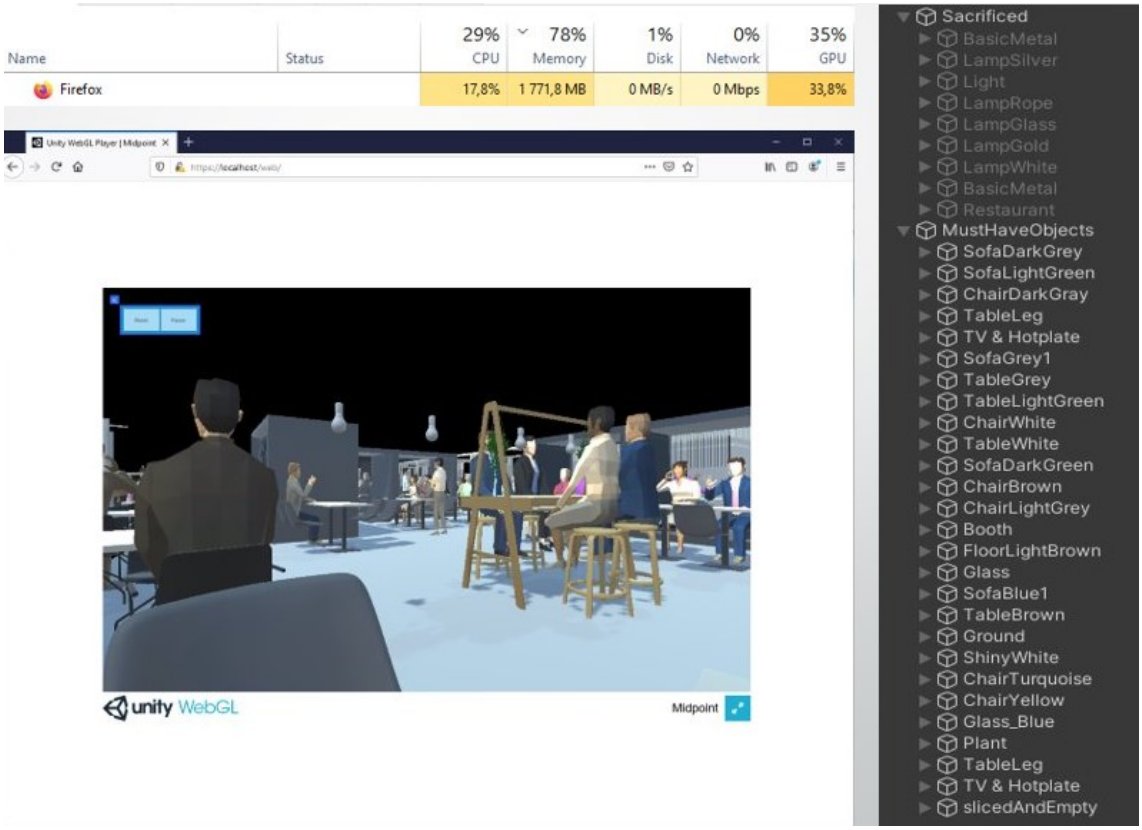
Esimerkkikoodi 9. Unity-editor-skripti, joka ryhmittää valitun peliobjektin jokaisen lapsiobjektin yhteisten materiaalien perusteella.

Lapsiobjektit ryhmittävän editor-skriptin avulla saatiin ryhmitettyä jokainen peliobjekti sitä vastaavan materiaalin hierarkiaan. Materiaaleja oli kokonaisuudessaan 35, mikä oli vain pieni osa objektien lukumäärään verrattuna, joten materiaalin perusteella ryhmittämisestä ei ollut merkittävää hyötyä. Tällä tapaa saatiin kuitenkin kokeiltua, miten eri objektiryhmien poistaminen maisemasta vaikuttaisi resurssien käyttöön ja pelaajan käyttökokemukseen.



Kuva 9. Unity-projektin rajattu näkymä, jossa kaikki peliobjektit on ryhmitetty materiaalin perusteella.

Maisemasta kokeiltiin poistaa eri materiaaliryhmiä, ja kaikkein paras tulos säilyttämällä samalla edes jollain tasolla uskottavuutta saatiin poistamalla lähes kaikki metallit ja lamput maisemasta. Niiden lisäksi ravintolan ulkokuoren todettiin vievän jo itsessään 2 GB muistia, joten sekin poistettiin. Näin saatiin vähennettyä WebGL-version muistin käyttö 5 GB:stä alle 2 GB:n. Kuvassa 10 nähdään optimoinnin jälkeisen maiseman lopputulos sekä lista poistetuista peliobjekteista.



Kuva 10. Selaimessa suoritettu optimoitu maisema.

WebGL-versioiden optimoinnin jälkeen todettiin, että ravintolasta oli jouduttu poistamaan liikaa informaatiota. Esimerkiksi kuvassa 10 näkyvä harmaapaitainen animoitu ihmismalli istuu ilmassa leijuvassa tuolissa eivätkä kuvassa näkyvät lamputkaan näyttäisi olevan kiinni missään. Tämä ei ollut haluttu lopputulos, joten WebGL-versioiden sijaan täytyi kehittää jotain muuta.

Vaikka WebGL-versiot olivat selvä epäonnistuminen, saatiin niistä arvokas opetus suurten 3D-mallien tuomista haasteista. Tämän lisäksi projekti koostui uudelleenkäytettävistä toimiviksi todetuista komponenteista.

6.2 Natiiviversiot ja niihin tarvittavat lisäykset

Koska insinöörityön tavoitteena oli alustariippumattoman interaktiivisen moninpelimuotoisen kokemuksen tarjoaminen ja lopputuloksen piti olla integroitavissa MyCampus-sovellukseen, täytyi projektia lähestyä uudella tapaa. WebGL-alustan sijaan projekti päätettiin rakentaa natiivisovelluksiksi Windows-, Android- ja Oculus Quest -alustoille, jotka olisivat ladattavissa MyCampus-sovelluksesta.

Windows-versio oli käytännössä valmis jo tähän mennessä tehdyillä komponenteilla sellaisenaan, mutta Android- ja Oculus-alustat vaativat omat lisäykset liikkumislogiikalle, ja sen lisäksi Oculus-alustalle piti myös ladata sitä tukeva SDK (Software Development Kit). SDK:ksi valittiin Unityn tarjoama XR Interaction Toolkit -ohjelmistokehys, joka mahdollistaa XR-kehityksen monelle eri alustalle samalla lähdekoodilla. XR Interaction Toolkit -ohjelmistokehykseen viitataan jatkossa lyhenteellä XRI.

Vaikka Windows-versio ei tarvinnut toimiakseen mitään ylimääräistä, päätettiin aihioprojektissa luotu pelaajan liikkumisskripti muuttaa sellaiseen muotoon, jossa liikkumissyötteessä käytetään tarkkailija-suunnittelumallia (observer pattern). Tämä tehtiin sen vuoksi, että samaa koodia voisi käyttää muiden natiivialustojen liikkumislogiikassa. Skriptille luotiin tarvittavat rajapintaluokat sekä staattinen luokka, joka pitää sisällään listan kaikista observereista. (Esimerkkikoodi 10.)

```
using System.Collections.Generic;
using System.Linq;
using UnityEngine;

public interface IEventObserver {}

public static class EventObservers {

    private static readonly List<IPlayerMovementInputObserver>
        movementObservers = new List<IPlayerMovementInputObserver>();

    private static readonly List<IDataInitializeObserver>
        dataInitObservers = new List<IDataInitializeObserver>();

    // -> Iterate all observer lists and register to
    //    an appropriate list if not already in it
    public static void Register(IEventObserver observer) {
        if (observer is IPlayerMovementInputObserver &&
            !movementObservers.Contains(observer)) {
                movementObservers.Add(
                    observer as IPlayerMovementInputObserver);
        }

        if (observer is IDataInitializeObserver &&
            !dataInitObservers.Contains(observer)) {
                dataInitObservers.Add(
                    observer as IDataInitializeObserver);
        }
    }

    // -> Iterate all observer lists and remove
    //    from an appropriate list if in it
    public static void Deregister(IEventObserver observer) {
        if (observer is IPlayerMovementInputObserver &&
            movementObservers.Contains(observer)) {
               movementObservers.Remove(
                   observer as IPlayerMovementInputObserver);
        }

        if (observer is IDataInitializeObserver
            && dataInitObservers.Contains(observer)) {
                dataInitObservers.Remove(
                    observer as IDataInitializeObserver);
        }
    }

    public static class Movement {
        public static void NotifyPlayerMoveForward()
            => movementObservers.ForEach(
                    (IPlayerMovementInputObserver observer)
                        => observer.OnPlayerMoveForward());

        public static void NotifyPlayerMoveBackwards()
            => movementObservers.ForEach(
                    (IPlayerMovementInputObserver observer)
                        => observer.OnPlayerMoveBackwards())
;
        public static void NotifyPlayerRotateRight()
            => movementObservers.ForEach(
                    (IPlayerMovementInputObserver observer)
```

```
                        => observer.OnPlayerRotateRight());

        public static void NotifyPlayerRotateLeft()
            => movementObservers.ForEach(
                    (IPlayerMovementInputObserver observer)
                        => observer.OnPlayerRotateLeft());

        public static void NotifyPlayerStrafeLeft()
            => movementObservers.ForEach(
                    (IPlayerMovementInputObserver observer)
                        => observer.OnPlayerStrafeLeft());

        public static void NotifyPlayerStrafeRight()
            => movementObservers.ForEach(
                    (IPlayerMovementInputObserver observer)
                        => observer.OnPlayerStrafeRight());
    }

    public static class Init {
        public static void NotifyPlayerPrefabInitialized(GameObject p)
            => dataInitObservers.ForEach(
                    (IDataInitializeObserver observer)
                        => observer.OnPlayerPrefabInstantiated(p));
    }
}
```

Esimerkkikoodi 10. Staattinen EventObservers-luokka, johon komponentit rekisteröityvät seuraamaan pelitapahtumia, kuten pelaajan syötettä.

Observer pattern -pohjan lisäksi luotiin uusi liikkumissyötettä seuraava moduuli, joka lähettää liikkumiskomennot staattiseen EventObservers-luokkaan rekisteröityneille observer-komponenteille (esimerkkikoodi 11). Observer patternin toimivuus testattiin Windows-versiossa.

```
using UnityEngine;
using UnityEngine.InputSystem;

public class RegularInputModule : MonoBehaviour
{

    void Update()
    {
        HandleMovementInputs();
    }

    private void HandleMovementInputs() {
        // rotate left
        if (Keyboard.current[Key.Q].isPressed) {
            EventObservers.Movement.NotifyPlayerRotateLeft();
        }

        // rotate right
        if (Keyboard.current[Key.E].isPressed) {
            EventObservers.Movement.NotifyPlayerRotateRight();
        }

        // move forward
        if (Keyboard.current[Key.W].isPressed) {
            EventObservers.Movement.NotifyPlayerMoveForward();
        }

        // move backwards
        if (Keyboard.current[Key.S].isPressed) {
            EventObservers.Movement.NotifyPlayerMoveBackwards();
        }

        // strafe left
        if (Keyboard.current[Key.A].isPressed) {
            EventObservers.Movement.NotifyPlayerStrafeLeft();
        }

        // strafe right
        if (Keyboard.current[Key.D].isPressed) {
            EventObservers.Movement.NotifyPlayerStrafeRight();
        }
    }
}
```

Esimerkkikoodi 11. RegularInputModule-skripti, joka välittää näppäimistöstä saadun syötteen perusteella liikkumiskomennon staattiselle EventObservers-observer-luokalle.

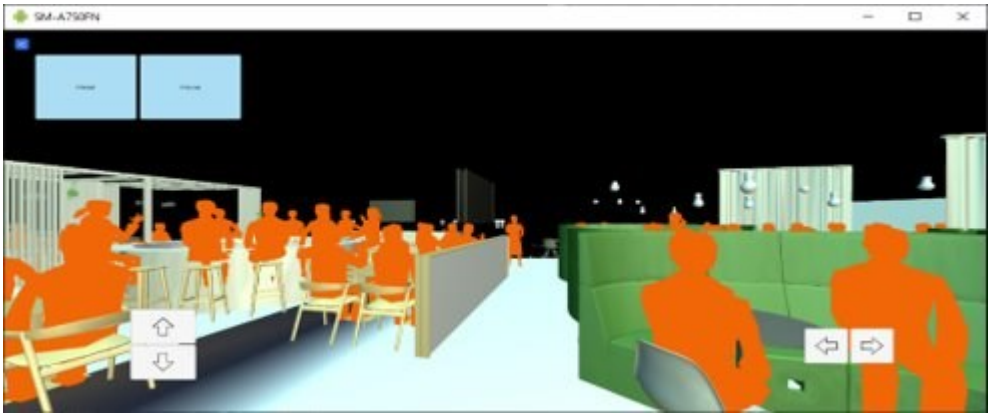
Android-versiolle tehtiin ravintolan 3D-mallin pohjalta oma näkymä, ja sille luotiin mobiililaitteita tukeva liikkumisskripti (esimerkkikoodi 12).

```
// Events sent from GUI buttons in MobileHUD are handled as movement
actions.

public class MobileInputModule : MonoBehaviour {

    // Booleans handled by button Event Triggers
    private bool movingForward = false;
    private bool movingBackwards = false;
    private bool movingLeft = false;
    private bool movingRight = false;

    void Update() {
        HandleMovement();
    }

    public void StartMovingForward() => movingForward = true;
    public void StartMovingBackwards() => movingBackwards = true;
    public void StartMovingLeft() => movingLeft = true;
    public void StartMovingRight() => movingRight = true;
    public void StopMovingForward() => movingForward = false;
    public void StopMovingBackwards() => movingBackwards = false;
    public void StopMovingLeft() => movingLeft = false;
    public void StopMovingRight() => movingRight = false;

    private void MoveLeft() {
        EventObservers.Movement.NotifyPlayerRotateLeft();
    }

    private void MoveRight() {
        EventObservers.Movement.NotifyPlayerRotateRight();
    }

    private void MoveForward() {
        EventObservers.Movement.NotifyPlayerMoveForward();
    }

    private void MoveBackwards() {
        EventObservers.Movement.NotifyPlayerMoveBackwards();
    }

    private void RotateLeft() {
        EventObservers.Movement.NotifyPlayerRotateLeft();
    }

    private void RotateRight() {
        EventObservers.Movement.NotifyPlayerRotateRight();
    }

    private void HandleMovement() {
       if (movingForward) MoveForward();
       if (movingBackwards) MoveBackwards();
       if (movingLeft) MoveLeft();
       if (movingRight) MoveRight();
    }
}
```

Esimerkkikoodi 12. MobileInputModule-skripti, joka RegularInputModule-skriptin (esimerkkikoodi 11) tavoin välittää liikkumiskomennot staattiselle EventObservers-observer-luokalle.

MobileInputModule-skriptissä kokeiltiin aluksi pyyhkäisysyötteen konvertoimista liikkumismuotoon, mutta sitä ei saatu mobiililaitteella toimimaan halutulla tavalla, joten sitä varten kehitettiin yksinkertainen graafinen käyttöliittymä, johon sijoitettiin napit sekä eteen- ja taaksepäin liikkumiselle että kääntymiselle (kuva 11).

Android-version suorittaminen ei optimoimattomalla 3D-mallilla onnistunut, vaan mallin koko aiheutti samoja ongelmia kuin WebGL-versioissakin. Vaikka Android-versio todettiinkin jälkeenpäin toimivaksi käyttämällä WebGL-versioiden optimoitua 3D-mallia mobiililaitteelle optimoituna (kuva 11), päätettiin, että mobiiliversiot jäisivät tältä erää WebGL-versioiden tavoin taustalle odottamaan jatkokehitystä. Sen sijaan alettiin keskittyä vain Oculus Quest- ja Windows-alustojen tukemiseen ja pyrittiin toteuttamaan insinöörityölle asetettuja tavoitteita.

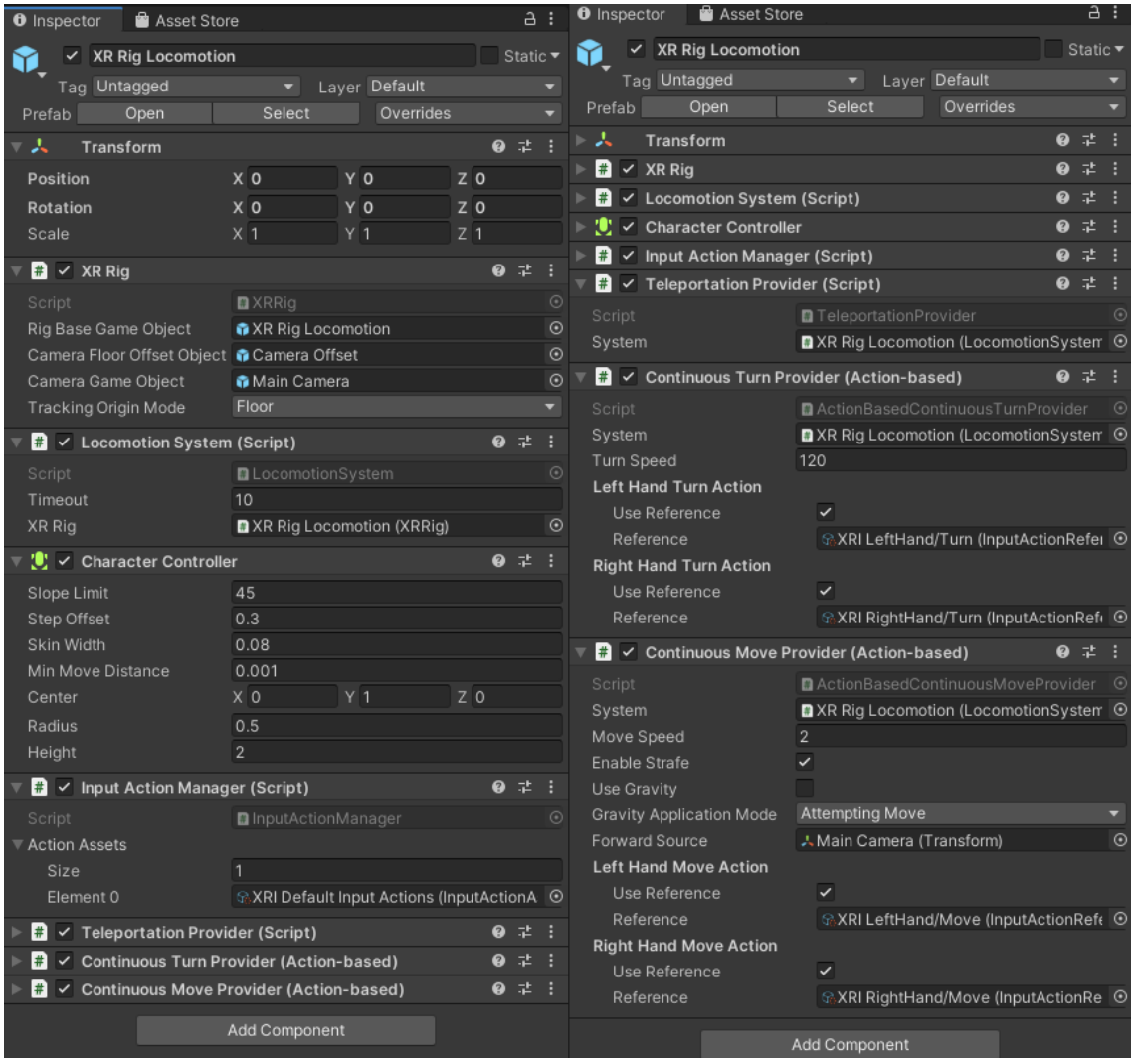


Kuva 11. Android-version suorittaminen fyysisellä laitteella.

Oculus-versiolle tehtiin Android-version tavoin oma näkymä, ja tarvittavan XRI-SDK:n lataamisen ja asentamisen jälkeen maisemaan luotiin XRI:n tarjoama XR Rig -peliobjekti, joka pitää sisällään sekä maiseman renderöimisen fyysiselle VR-laitteelle ja fyysisen laitteen seuraamista mahdollistavan virtuaalisen kameran että fyysisten ohjainten syötteitä lukevat ja niiden sijaintia seuraavat virtuaaliset VR-ohjaimet (kuva 12).

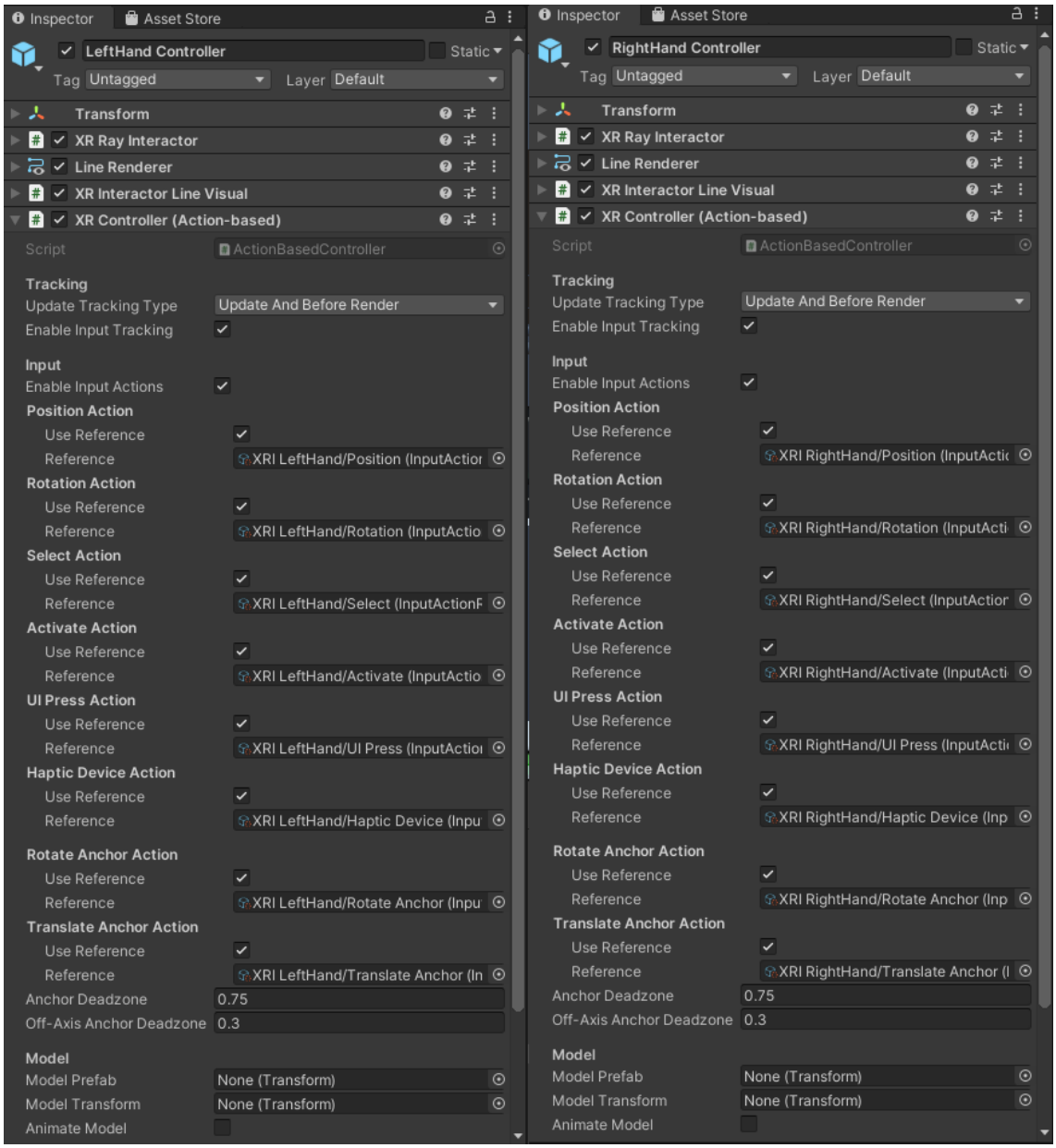
Kuva 12. XRI:n tarjoama XR Rig -peliobjekti.

XR Rig -peliobjekti muutettiin XR Rig Locomotion -nimiseksi, ja sille asetettiin XRI:n sisäänrakennetut VR-laitteella liikkumisen mahdollistavat skriptit (kuva 13).



Kuva 13. XR Rig -peliobjektiin lisätyt VR-laitteella eri tavoin liikkumisen mahdollistavat skriptit.

Kuvassa 13 nähtävät skriptit mahdollistavat sekä ohjaimella liikkumisen että kääntymisen vastaavien syötteiden tapahtuessa. Syötteiden laukaisutapahtumat määritettiin XR Rig -peliobjektiin sisäänrakennettujen ohjaimien sisäänrakennetuista skripteistä, jotka asetettiin automaattisesti XRI:sta saatavan Default Input Actions -moduulin kautta (kuva 14).



Kuva 14. XR Rig -ohjaimien syötteiden referenssit.

VR-versiota testattiin sekä natiivina Oculus Quest 2:lla että PC:llä renderöitävänä Oculus Rift -ohjelman kautta. Optimoimaton maisema oli natiivina liian raskas Oculus Quest 2:n suoritettavaksi, joten testaamisessa jouduttiin käyttämään samaa mobiililaitteelle optimoitua maisemaa kuin Android-versiossa. Oculus Questin tukeminen natiivina jouduttiin Androidin tavoin sulkemaan tältä erää ulos projektista ja tuetuiksi natiivisovellusalustoiksi valittiin Windows- ja PC VR -alustat.

Natiivisovelluksien avulla saatiin toteutettua moninpelimuotoinen interaktiivisuus, mutta pelkkien latauslinkkien tuomista MyCampus-sovellukseen ei todettu riittäväksi, joten natiivisovelluksien lisäksi päätettiin, että loppusovellukseen integroitava osuus tulisi olemaan selaimessa nähtävä reaaliaikainen suoratoisto virtuaalisen ravintolan näkymästä.

6.3 FFmpegin kokeileminen

Unity-projektin reaaliaikaisen suoratoiston toteuttamisen ratkaisujen etsimiseen käytettiin runsaasti aikaa, mutta aiheesta löytyi erittäin vähän tietoa. Aluksi tutustuttiin Unityn tarjoamaan WebRTC-esittelyprojektiin (Web Real Time Communication), jonka avulla todettiin, että suoratoistettu ratkaisu tulisi olemaan toimiva, sillä ravintolan sai reaaliaikaisesti suoratoistettua sekä selaimeen että mobiililaitteelle erittäin vähäisellä resurssien käytöllä. Suoratoistettu kuva oli sen lisäksi erittäin hyvälaatuista.

Vaikka WebRTC-esittelyprojektin pohjan käyttäminen ravintolaprojektissa todettiin toimivaksi, se oli toteutettu aivan liian monimutkaisesti ravintolaprojektin tarpeisiin nähden, eikä sitä ollut suunniteltu tuotantoon. Tämän vuoksi alettiin tutkia toista lupaavalta vaikuttavaa median suoratoiston mahdollistavaa FFmpeg-multimediatyökalua sekä reaaliaikaisen suoratoiston palvelemisen mahdollistavaa NGINX-RTMP-moduulia.

FFmpegin tutkimisen ja lataamisen jälkeen kokeiltiin, miten se toimii käytännössä. Tämä aloitettiin kokeilemalla kopioida olemassa oleva video eri säiliömuotoon toiseen tiedostoon. Prosessi vaati vain yhden lyhyen komentorivillä suoritettavan komennon (esimerkkikoodi 13).

```
$ ffmpeg -i testvideo.mov newvideo.mp4
```

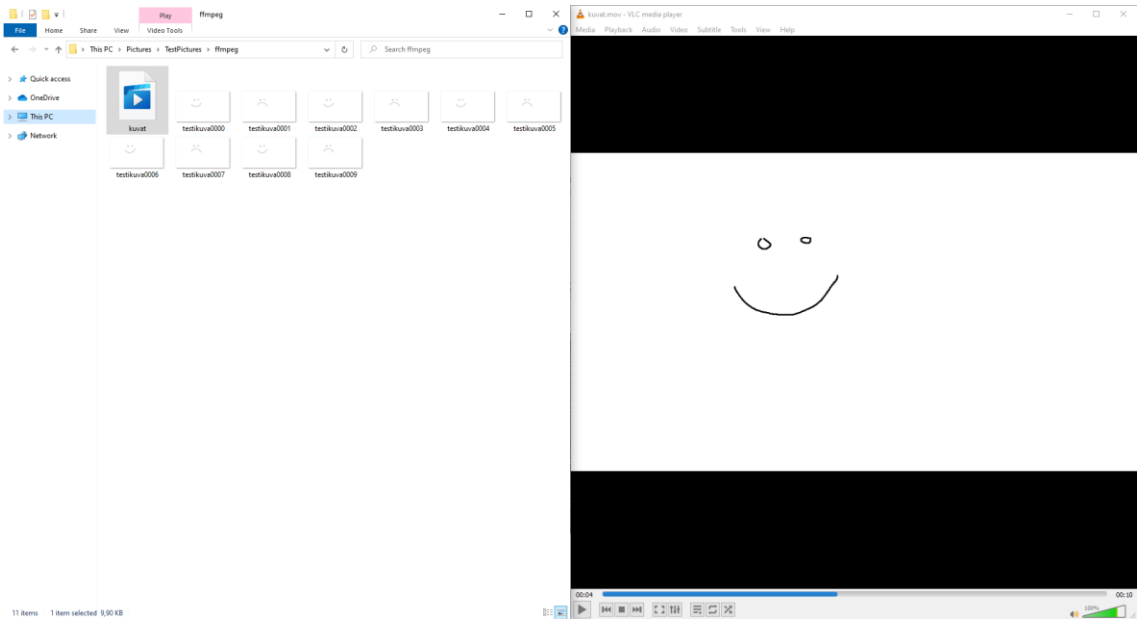
Esimerkkikoodi 13. Tiedoston testvideo.mov kopioiminen newvideo.mp4-tiedostoon FFmpegin avulla.

Videon kopioiminen FFmpegilla uuteen tiedostoon oli erittäin yksinkertaista, mutta itse tehtäväkään ei ollut kovin vaativa, joten seuraavaksi kokeiltiin, miten lukuisat kuvat kopioitaisiin videoksi. Tämän toteuttamiseen luotiin 10 kuvaa, jotka kopioitiin löydetyn esimerkkikomennon (esimerkkikoodi 14) avulla videoksi. Jotta esimerkkikomento toimisi, piti kaikki kuvat nimetä esimerkkikomennossa esiintyvän säännöllisen lausekkeen mukaisesti.

```
$ ffmpeg -f image2 -framerate 1 -i "testikuva%04d.png" kuvat.mov
```

Esimerkkikoodi 14. FFmpeg-komento, joka kopioi kansiossa sijaitsevat testikuva-nimellä alkavat ja neljään numeroon päättyvät PNG-tiedostomuotoiset kuvat kuvat.mov-nimiseksi videotiedostoksi, jossa jokainen ruutu on yhden sekunnin pituinen eli jokainen kuva näkyy videolla yhden sekunnin ajan.

Kuvat saatiin onnistuneesti kopioitua videoksi (kuva 15), joten samaa käytäntöä päätettiin soveltaa myös Unity-projektissa.



Kuva 15. Kuvatiedostot videoksi kopioivan FFmpeg-komennon havainnollistettu tulos.

Virtuaalisen ravintolaprojektin reaaliaikaista suoratoistoa varten luotiin uusi Unity-projekti, jossa käytettiin samaa optimoimatonta pohjaprojektia kuin Windows- ja PC VR -alustoille rakennettavassa projektissa. Projektiin luotiin skripti, joka ottaa jokaisella ruudulla kuvan ja tallentaa sen PNG-muotoon (esimerkkikoodi 15).

```
using UnityEngine;
using System.IO;

public class ScreenshotController: MonoBehaviour
{
    private string screenshotsDir;

    private int frameNum;

    private void Awake() {
        CreatePaths();
    }

    void Update()
    {
        SaveScreenShot();
    }

    // Create a media/screenshots directory for saving screenshots
    private void CreatePaths() {
        string currentDir = Directory.GetCurrentDirectory();
        string mediaDir = currentDir + "/Media";
        screenshotsDir = mediaDir + "/screenshots";
        Directory.CreateDirectory(screenshotsDir);
    }

    private void SaveScreenShot() {
        string fileName = screenshotsDir
            + "/midpoint"
            + PushZeroRecursive(frameNum.ToString(), 4)
            + ".png";

        ScreenCapture.CaptureScreenshot(fileName);

        frameNum++;
    }

    // Push zeroes to a string formatted number until it's in the
form: XXXX
    private string PushZeroRecursive(string numStr, int suffixLen) {
        return (numStr.Length >= suffixLen)? numStr :
            PushZeroRecursive("0" + numStr, suffixLen);
    }
}
```

Esimerkkikoodi 15. Unity-projektista jokaisen ruudun midpoint-nimellä alkavaksi neljään numeroon päättyväksi PNG-muotoiseksi uudeksi tiedostoksi tallentava skripti.

Kuvat kopioitiin videoksi aiemmin toimivaksi todetulla FFmpeg-komennolla (esimerkkikoodi 14). Komento muutettiin yhden kuvan sijaan kopioimaan sekunnin aikana 24 kuvaa, ja myös Unity asetettiin tallentamaan saman verran ruutuja sekunnin aikana. Tuotetun videon pystyi toistamaan odotetulla tavalla onnistuneesti. Komennon testaamisen jälkeen Unity-projektin annettiin olla 5 minuuttia

suoritettavana, jotta ravintolasta saatiin 5 minuutin pituinen video suoratoiston kokeilua varten. Reaaliaikaisen suoratoiston palvelintoteutusten tutkimisen jälkeen valittiin kokeiltavaksi NGINX-RTMP-moduuli.

6.4 Suoratoistopalvelin ja sen kontittaminen

Suoratoistopalvelin luotiin NGINX:n virallisen RTMP-moduuli-tutoriaalin pohjalta (Enabling Video Streaming for Remote Learning with NGINX and NGINX Plus). Palvelimessa käytettiin RTMP-moduuliin sisäänrakennettua NGINX-konfiguraatiotiedostoa, jossa paljastetun 1935-portin sisään tuleva RTMP-tietoliikenne muunnetaan HLS-soittolistoiksi sekä paljastetaan portti 80, josta HLS-soittolistaa palvellaan (esimerkkikoodi 16). Tässä oli käytännössä kaikki, mitä suoratoistopalvelimen toteuttaminen vaati toimiakseen.

```
user www-data;
worker_processes auto;
pid /run/nginx.pid;
include /etc/nginx/modules-enabled/*.conf;

events {
     worker_connections 1024;
}

rtmp {
    server {
        listen 1935;
        application live {
            live on;
            interleave on;
            hls on;
            hls_fragment 3;
            hls_playlist_length 60;
            hls_path /tmp/hls;
        }
    }
}

http {
    include       mime.types;
    default_type application/octet-stream;

    server {
        listen 80;

        location / {
            root /tmp;

            types {
                application/vnd.apple.mpegurl m3u8;
                video/mp2t ts;
                application/dash+xml mpd;
            }
        }

    }
}
```

Esimerkkikoodi 16. NGINX-RTMP-moduuliin sisäänrakennettu NGINX-konfiguraatiotiedosto suoran lähetyksen vastaanottamiseen ja palvelemiseen.

Palvelin pakattiin Dockerfile-tiedostolla (esimerkkikoodi 17) lokaalisti suoritettavaksi Docker-imageksi.

```
FROM ubuntu

ENV DEBIAN_FRONTEND=noninteractive

RUN apt-get update && \
    apt install nginx -y

COPY nginx.conf /etc/nginx/nginx.conf

RUN apt install libnginx-mod-rtmp -y

CMD ["nginx", "-g", "daemon off;"]
```

Esimerkkikoodi 17. NGINX-RTMP-moduulin Docker-imageksi pakkaava Dockerfile, jossa ladataan NGINX-RTMP-moduuli ja kopioidaan suoratoiston mahdollistava NGINX-konfiguraatiotiedosto Ubuntu-image-tiedostolle.

Kontitetun suoratoistopalvelimen toimivuus varmistettiin RTMP-protokollalla videota jatkuvasti localhost-IP-osoitteeseen lähettävällä FFmpeg-komennolla (esimerkkikoodi 18).

```
$ ffmpeg -re -i Demo.mp4 -vcodec copy -loop -1 -c:a aac -f flv
rtmp:127.0.0.1/live/demo
```

Esimerkkikoodi 18. RTMP-protokollalla lokaalille NGINX-palvelimelle videota lähettävä FFmpeg-komento.

Koska ennalta nauhoitetun videon lähettäminen lokaalille NGINX-palvelimelle todettiin toimivaksi, luotiin sen mahdollistavasta komennosta suoritettava shell-skripti. Sekä luotu shell-skripti että ennalta nauhoitettu video kontitettiin omalla Dockerfile-tiedostollaan. Tästä luotua Docker-image-tiedostoa päätettiin käyttää varasuunnitelmana, jos reaaliaikainen suoralähetys ravintolasta ei jostain syystä toimisi.

```
FROM ubuntu

ENV DEBIAN_FRONTEND=noninteractive

RUN apt-get update

RUN apt install ffmpeg -y && \
    mkdir server

COPY Demo.mp4 server/Demo.mp4

COPY stream-unity.sh stream-unity.sh

RUN chmod 700 stream-unity.sh

ENTRYPOINT ./stream-unity.sh
```

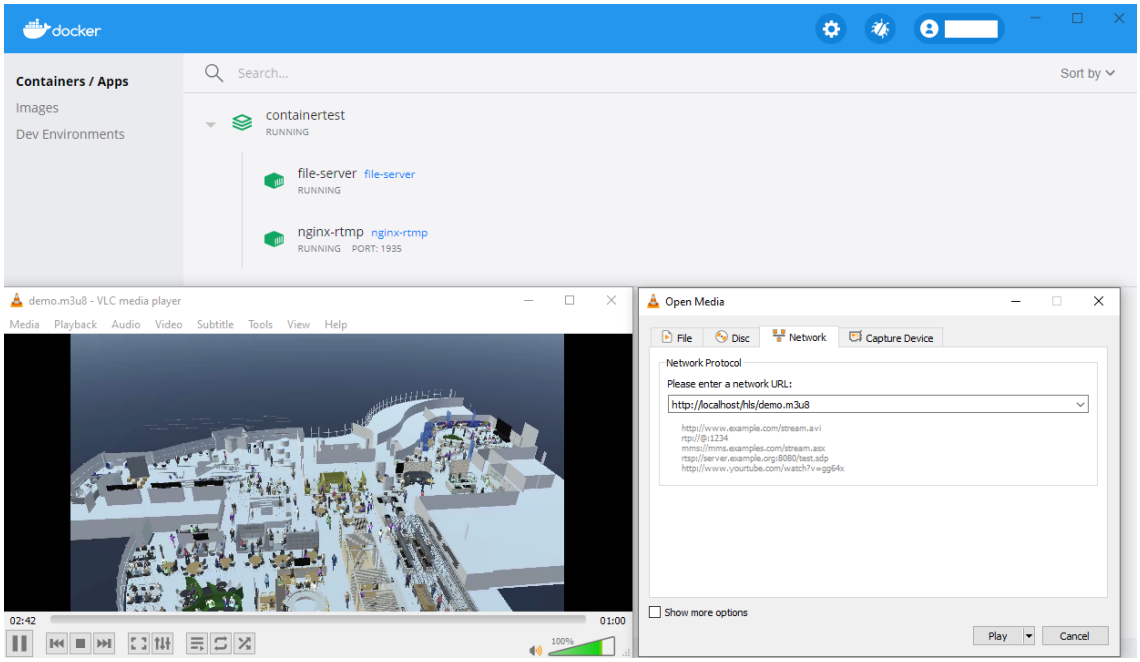
Esimerkkikoodi 19. Dockerfile-tiedosto, jossa Ubuntu-image-tiedoston sisään ladataan FFmpeg sekä kopioidaan Demo.mp4-videotiedosto ja sitä FFmpeg-komennolla (esimerkkikoodi 18) lähettävä stream-unity.sh-skripti. Shell-skriptille annetaan lisäksi suoritusoikeudet, jotta sitä voidaan suorittaa kontin sisällä.

Sekä NGINX-suoratoistopalvelimen että Demo.mp4-videota lähettävän kontitetun sovelluksen Docker-image-tiedostot liitettiin yhteen Docker-compose-tiedoston avulla (esimerkkikoodi 20).

```
services:

  nginx:
    image: nginx-rtmp
    container_name: nginx-rtmp
    build:
      context: ./nginx-rtmp
    depends_on:
      - file-server
    ports:
      - 80:80
      - 443:443
      - 1935:1935
    restart: on-failure

  file-server:
    image: file-server
    container_name: file-server
    build:
      context: ./file-server
    restart: on-failure
```

Esimerkkikoodi 20. Ravintolaprojektista nauhoitetun demovideon suoralähettämisen yhdellä komennolla mahdollistava docker-compose-tiedosto.

Esimerkkikoodissa 20 esitetyssä docker-compose-tiedostossa luodaan nginx- ja file-server-nimiset virheen tapahtuessa automaattisesti uudelleenkäynnistettävät palvelut, joiden context-arvoissa määritetään, mistä kansioista Docker-image-tiedostot rakennetaan. Nginx-palvelun context-arvo viittaa kansioon, jossa aiemmin luotu NGINX-RTMP-moduulia käyttävä Dockerfile (esimerkkikoodi 17) sijaitsee, ja file-server-palvelun context-arvo viittaa kansioon, jossa virtuaalisen ravintolan demovideon lokaalille NGINX-palvelimelle lähettävä Dockerfile-tiedosto (esimerkkikoodi 19) sijaitsee. Docker-compose-build- ja docker-compose-up-komentojen tulos nähdään kuvassa 16.

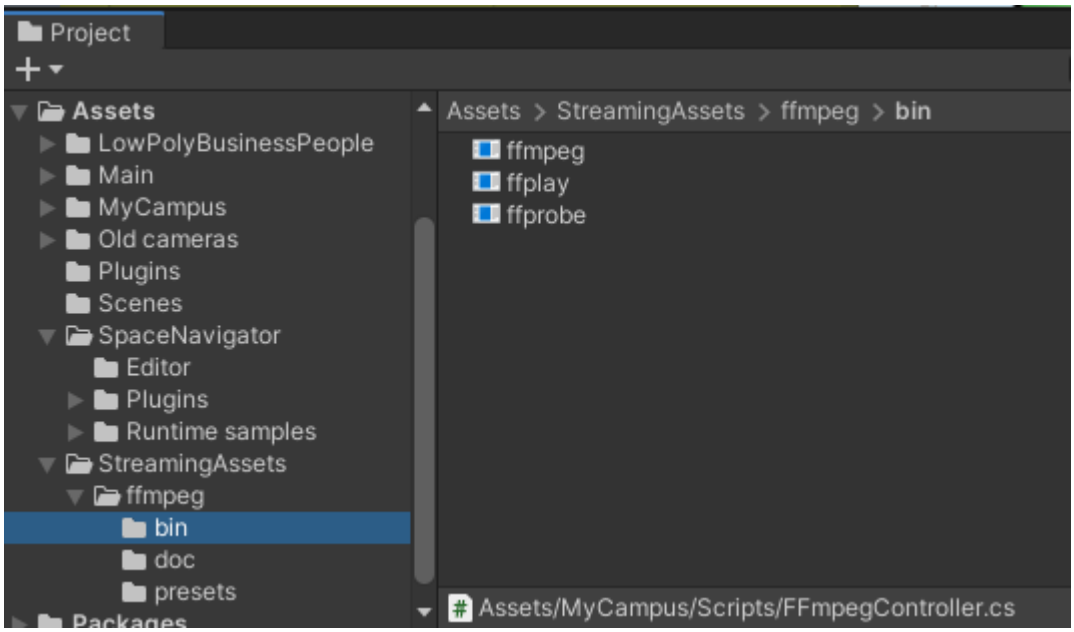


Kuva 16. Esimerkkikoodi 20:ssa kuvatun docker-compose-tiedoston suorittaminen ja VLC-mediasoittimella suoratoistetun videon hakeminen NGINX-palvelimelta.

Tähän mennessä oli saatu tehtyä toimiva kontitettu suoratoistopalvelin. Virtuaalista ravintolaa oli kokeiltu suoratoistaa ennalta nauhoitetun videon kautta onnistuneesti, mutta se ei ollut insinöörityön tavoitteiden kannalta tarpeeksi, vaan ravintolaa oli saatava suoratoistettua sekä reaaliaikeisesti että julkaisuvaiheessa jossain määrin automatisoidusti. Seuraavaksi kokeiltiin, miten FFmpegin saisi sulautettua Unity-projektiin ja siitä rakennettuihin versioihin.

6.5 FFmpegin sulauttaminen Unity-projektiin

Jotta Unity-versioita saataisiin suoratoistettua ilman sekä FFmpeg-riippuvuuksien lataamista että erillisen skriptin suorittamista jokaisen julkaistun version yhteydessä, tuotiin FFmpeg-riippuvuudet Unity-projektin sisään. Tämä mahdollisti sen, että FFmpeg-komentoja pystyi käyttämään suoraan Unity-skripteistä viittaamalla FFmpeg-kansiossa sijaitsevaan ffmpeg.exe-ohjelmatiedostoon. Jotta FFmpeg-riippuvuudet saatiin rakennettua jokaisen version sisään, ne piti sijoittaa Assets-kansioon luotuun StreamingAssets-kansioon (kuva 17). Unity kopioi oletuksena StreamingAssets-kansion sisällön rakennettuihin versioihin.



Kuva 17. StreamingAssets-kansion sisään tuodut FFmpeg-riippuvuudet.

Kun FFmpeg-riippuvuudet oli tuotu Unity-projektiin (kuva 17), FFmpegin käyttäminen onnistui luomalla ffmpeg.exe-ohjelmatiedostoon viittaava prosessi, jolle syötettiin tarvittava komento argumentteina (esimerkkikoodi 21).

```
private void FFMpegWithArgs(string args, EventHandler eventHandler) {
        try {
            var task = Task.Run(() => {
                UnityEngine.Debug.Log("Starting FFmpeg...");

                ffmpeg = new Process();

                ffmpeg.EnableRaisingEvents = true;

                ffmpeg.StartInfo.FileName = ffPath;
                ffmpeg.StartInfo.UseShellExecute = false;
                ffmpeg.StartInfo.CreateNoWindow = true;
                ffmpeg.StartInfo.Arguments = args;
                ffmpeg.StartInfo.RedirectStandardError = false;

                ffmpeg.Exited += eventHandler;

                ffmpeg.Start();

                UnityEngine.Debug.Log("FFmped started with args: " +
                                      ffmpeg.StartInfo.Arguments);
            });
        }
        catch (Exception e) {
            UnityEngine.Debug.Log(e);
        }
    }
```

Esimerkkikoodi 21. FFmpegWithArgs-funktio, joka luo taustalla säikeessä suoritettavan ikkunattoman FFmpeg-prosessin, joka suorittaa funktioon syötetyt argumentit. Prosessin poistumisfunktioon lisätään EventHandler-tyyppinen takaisinkutsufunktio, jonka avulla saadaan tieto siitä, että prosessi on ohi.

Aiemmin tehty kuvakaappaus-skripti (esimerkkikoodi 15) muutettiin sellaiseen muotoon, että kuvista koostettiin 240 ruudun välein videoklippi (esimerkkikoodi 22) FFmpegWithArgs-funktiota (esimerkkikoodi 21) käyttäen. Skriptissä asetettiin sovelluksen kuvataajuus 24:ään, joten kerralla luotiin 10 sekunnin videoklippejä. Videoklippien luomisen lisäksi poistettiin kuvia sitä mukaa, kuin niitä käytettiin.

```
void Update()
{
    SaveScreenShot();

    if (frameNum % 240 == 0)
    {
        ConvertMP4AndDelete();
    }
}

private async void ConvertMP4AndDelete()
{
    CreateClip(lastIndex, framesPerClip);

    // Wait 5 seconds, then delete used images
    await Task.Delay(5000);

    for (int i = 0; i < (framesPerClip); i++)
    {
        // Delete file
        File.Delete(framePaths[0]);
        // Delete filename reference
        framePaths.RemoveAt(0);
    }
}

private void CreateClip(int startIdx, int numOfFrames)
{
    EventHandler eventHandler = new EventHandler(FFmpegExit);

    string startNumArg = PushZeroRecursive(startIdx.ToString(), 4);
    string args =
        $"-start_number {startNumArg} " +     // Frame to start from
        $"-i {framesDir}/midpoint%04d.png " + // Input: all .png's
                                                 that end in 4 digits
        $"-vframes {numOfFrames} " +          // How many frames
        $"{clipsDir}/midpoint_{startIdx}
          {startIdx + numOfFrames - 1}.mp4"; // Output mp4

    FFMpegWithArgs(args, eventHandler);

    lastIndex = startIdx + framesPerClip;
}
```

Esimerkkikoodi 22. Unity-skriptin Update-, ConvertMp4AndDelete- ja CreateClip-funktiot, jotka luovat yhdessä videoklipin tallennetuista PNG-tiedostoista aina 240 ruudun välein.

Renderöity kuva Unity-projektista oli tässä vaiheessa onnistuttu tallentamaan videoklipeiksi reaaliajassa, mutta kuva piti vielä lähettää NGINX-palvelimelle suoratoistettavaksi.

Vaikka videoklippien yhdistäminen yhdeksi videoksi oli mahdollista suhteellisen yksinkertaisella FFmpeg-komennolla, niiden yhdistäminen jatkuvaksi lähetykseksi oli päinvastaista. Tämän lisäksi jatkuva kuvien tallentaminen ja poistaminen vei liikaa resursseja. Ongelman tutkimiseen käytettiin runsaasti aikaa, minkä jälkeen päätettiin yrittää löytää toinen tapa kuvan lähetykselle kuin kuvien tallentaminen. Ratkaisuksi luotiin RenderTexture-tiedosto, johon voi renderöidä Unity-maisemassa sijaitsevan kameran havaitseman kuvan. RenderTexture-tiedoston käyttäminen poisti tarpeen jokaisen ruudun tallentamisprosessille.

RenderTexture-tiedoston lähettäminen FFmpeg-prosessille tehtiin Unity-projektiin luodun UDP-asiakasohjelman kautta (esimerkkikoodi 23).

```
using System;
using System.Net.Sockets;
using System.Threading.Tasks;
using UnityEngine;

class UDPClient
{
    private UdpClient client;

    public void Connect(int port) {
        Task.Run(() => {
            try {
                client = new UdpClient("127.0.0.1", port);
                Debug.Log("socket connected");
            }
            catch (Exception e) {
                Debug.Log(e);
            }
        });
    }
    public void Send(byte[] sendbuf) {
        try {
            client.Send(sendbuf, sendbuf.Length);
        } catch (Exception e) {
            Debug.Log(e);
        }
    }

    public void Disconnect() {
        client.Close();
    }
}
```

Esimerkkikoodi 23. Unity-projektiin luotu UDP-asiakasohjelma, jonka Send-funktio lähettää localhost-IP-osoitteeseen funktioon syötettyä tavutaulukkoa.

UDP-asiakasohjelmasta luotiin objekti FFmpegWithArgs-funktion sisältämään Unity-skriptiin (esimerkkikoodi 21). Samaan skriptiin tehtiin luodulle RenderTexture-tiedostolle muuttuja, johon asetettiin referenssi Unity-editorin kautta. Seuraavaksi luotiin tarvittavat funktiot sekä RenderTexture-tiedostoon renderöidyn kuvan tavutaulukon lukemiselle että sen UDP-asiakasohjelmalle lähettämiselle (esimerkkikoodi 24). Koska UDP-paketin maksimikoko on noin 65 kilotavua, jouduttiin RenderTexture-tiedostoon renderöidyn kuvan resoluutio asettamaan 736 x 414 pikseliin. Tämä arvo todettiin kokeilun kautta sellaiseksi maksimiarvoksi, jolla jokainen UDP-paketti saatiin lähetettyä. Resoluutiossa tavoiteltiin 1,78:1-kuvasuhdetta.

```
private void InitTexture() {
    texture = new Texture2D(
        renderTxt.width, renderTxt.height, TextureFormat.RGB24,false);
}

private void ReadTexture() {
    RenderTexture.active = renderTxt;
    texture.ReadPixels(
        new Rect(0, 0, renderTxt.width, renderTxt.height), 0, 0);

    texture.Apply();
}

private void SendTextureUDP() {
    try {
        ReadTexture();

        byte[] bytes = texture.EncodeToJPG();

        if (bytes.Length > 65527) {
            tooBigForUDPCount++;

            UnityEngine.Debug.LogError(
                "Packet too big for UDP. " +
                $"Total count of such packets: {tooBigForUDPCount}");
        } else {
            client.Send(bytes);
        }
    }catch (Exception e) {
        UnityEngine.Debug.Log(e);
    }
}
```

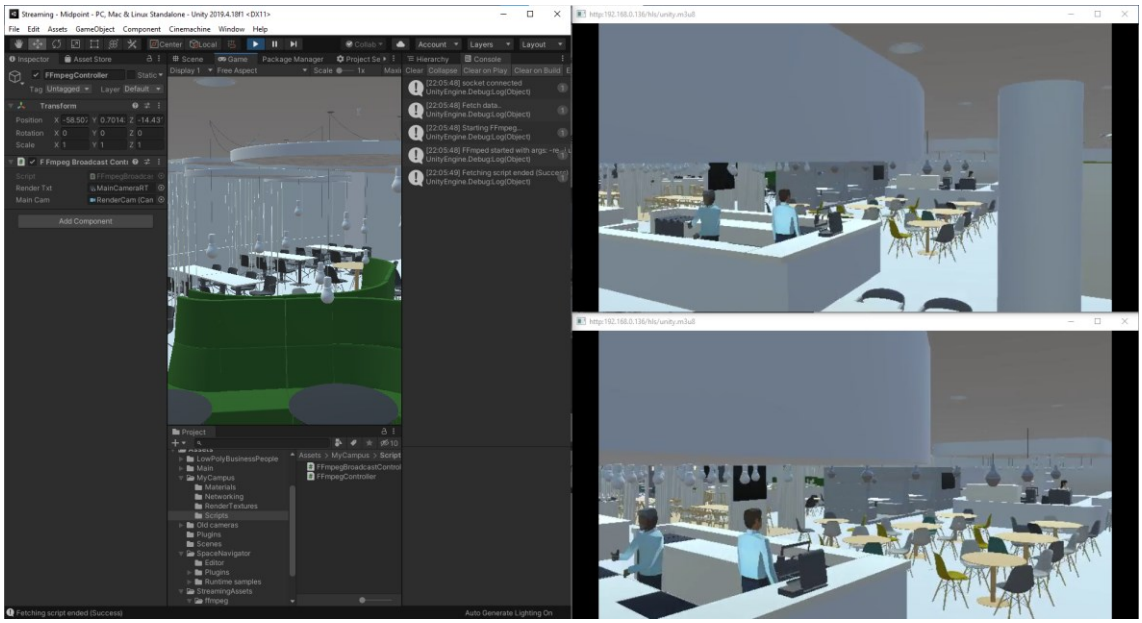
Esimerkkikoodi 24. Unity-skriptiin luodut InitTexture-, ReadTexture- ja SendTextureUDP-funktiot, jotka yhdessä alustavat RenderTexture-tiedoston, muuntavat siihen tallennetut pikselit tavutaulukoksi ja lähettävät tavutaulukon UDP-asiakasohjelman kautta localhost-IP-osoitteeseen skriptissä määritettyyn porttiin.

FFmpeg-prosessi aloitettiin skriptissä argumenteilla, joissa sisääntuloksi määritettiin UDP-asiakasohjelmalta suoratoistettu tavutaulukko ja ulostulo määritettiin lähettämään sisääntulosta saadun tavutaulukon h264-videokoodekilla pakattua äänetöntä FLV-tiedostomuotoista suoralähetystä RTMP-protokollaa käyttäen lähetystä vastaanottavalle lokaalille NGINX-palvelimelle (esimerkkikoodi 25).

```
private void StartFFmpeg() {
    // Read byte stream from UDP ->
    // encode with h264 & remove audio ->
    // output to rtmp stream
    string args = "-re -i udp://127.0.0.1:50500 -c:v libx264 -vprofile
                + " baseline -an -f flv " + nginxUrl;

    EventHandler handler = new EventHandler(FFmpegExit);

    FFMpegWithArgs(args, handler);
}
```

Esimerkkikoodi 25. StartFFmpeg-funktio, joka käynnistää FFmpeg-prosessin.

FFmpeg-prosessin aloittava skripti asetettiin maisemassa sijaitsevaan peliobjektiin, ja sitä kokeiltiin onnistuneesti (kuva 18).



Kuva 18. Käynnissä oleva suoratoistettua kuvaa virtuaalisesta ravintolasta lokaalissa verkossa sijaitsevalle NGINX-palvelimelle lähettävä Unity-projekti sekä palvelimelta saatua kuvaa suoratoistavat kaksi ffplay-mediatoistinikkunaa.

Unity-suoratoistoprojektista saatiin tuotettua 736 x 414 -resoluutioinen reaaliaikainen näkymä virtuaalisesta ravintolasta, josta fyysisen ravintolan käyttöastetta pystyi seuraamaan. Insinöörityön tärkein tavoite oli täten saatu toteutettua.

Suoratoistoprojekti koetettiin kontittaa, mutta graafisten Unity-versioiden suorittaminen ilman aktiivista monitoria tuotti sen verran ongelmia, että Unity-version suorittaminen manuaalisesti koettiin riittäväksi. Suoratoistoa jakava RTMP-palvelin julkaistiin MyCampus-Azure-ympäristöön, ja seuraavaksi lähdettiin toteuttamaan MyCampus-sovellukseen integroitavissa olevaa web-käyttöliittymää Reactilla.

6.6 Web-käyttöliittymän toteuttaminen Reactilla

Käyttöliittymä luotiin uuteen React-projektiin npm-komennolla npx-create-react-app, ja siihen asennettiin npm-paketinhallintajärjestelmän kautta HLS-soittolistan soittamisen mahdollistava react-player-komponentti. Tämän lisäksi asennettiin automaattisesti responsiivisuudesta huolehtiva valmiita tyylejä tarjoava material-ui-kirjasto. (Esimerkkikoodi 26.)

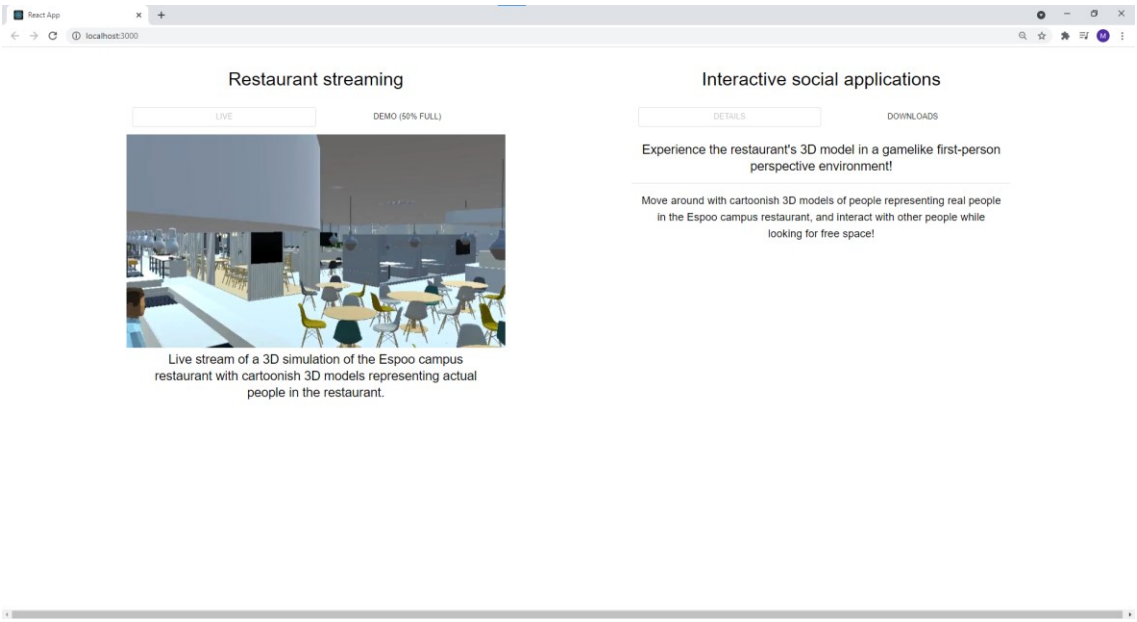
```
$ npx create-react-app
$ npm install react-player
$ npm install @material-ui/core
```

Esimerkkikoodi 26. React-käyttöliittymän luomiseen ja tarvittavien riippuvuuksien lataamiseen käytetyt npm-komennot.

Käyttöliittymään luotiin kaksi pääkomponenttia. Oleellisin näistä kahdesta komponentista oli StreamPlayer-komponentti, johon luotiin komponenttiin määritettyä web-osoitetta suoratoistava react-player-komponentti. StreamPlayer-komponentissa suoratoistettiin oletuksena luvussa 6.3 tehtyä ennalta nauhoitettua NGINX-palvelimelta haettua videota. StreamPlayer-komponentin lisäksi luotiin komponentti, josta pystyisi tulevaisuudessa lataamaan natiiviversioita.

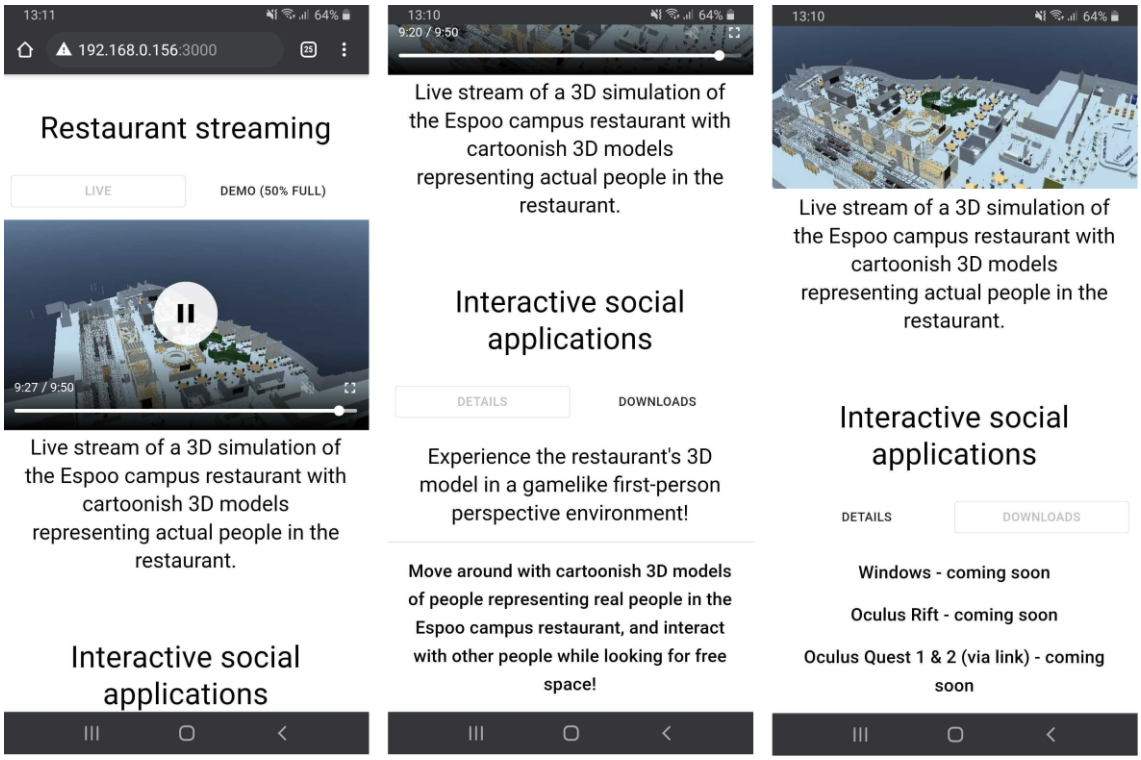
Tarjotusta web-sivusta tehtiin responsiivinen material-ui-kirjastosta saatujen Grid-komponenttien avulla. Grid-komponentteihin koettiin riittäväksi tehdä vain yksi pysäytyspiste 960 pikseliin. Molempien komponenttien sisältö keskitettiin.

Leveämmille kuvaruuduille (yli 960 pikseliä) asetettiin yksittäisten komponenttien leveydeksi neljä kahdestoistaosaa koko ruudun leveydestä ja molemmat komponentit sijoitettiin vierekkäin (kuva 19). Komponentit käyttivät siis yhdessä kahdeksan kahdestoistaosaa koko kuvaruudun leveydestä.



Kuva 19. Käynnissä oleva React-asiakasohjelman käyttöliittymä yli 960 pikseliä leveässä web-selaimessa.

Kapeampiruutuisille (alle 960 pikseliä) käyttöliittymille koettiin parhaaksi asettaa kumpikin komponentti käyttämään leveytenä kuvaruudun täyttä leveyttä (kuva 20).



Kuva 20. Kooste React-käyttöliittymän alle 960 pikseliä leveälle mobiililaitteelle renderöidyistä eri näkymistä.

Vaikka moninpelimuotoiset natiiviversiot jo sellaisinaan toimivat, niille koettiin parhaaksi jatkokehittää sekä paremmat käyttöliittymät että uusia toiminnallisuuksia, ennen kuin niille kehitettäisiin palvelin, josta ne olisivat ladattavissa.

Käyttöliittymää ja siihen liittyviä MyCampus-Azure-ympäristön palvelimia testattiin oma aikansa, minkä tuloksena todettiin sen ensimmäinen versio toimivaksi, joten insinöörityölle asetetut tavoitteet oli saatu toteutettua ja työ oli täten saatu päätökseen.

7 Yhteenveto

Raportin alussa käydyn virtualisointiin ja kontittamiseen liittyvän tutkimustyön pohjalta saatiin vankka perusta insinöörityön jokaisen työnvaiheen suunnittelulle ja toteutukselle. Tutkimustyössä käsitellyt virtuaalikoneet ja Docker-konttiteknologia olivat työosuudessa jatkuvasti esillä, ja ne tulevat jatkossa olemaan oleellinen osa tekijän jokapäiväisesti käyttämää taitotietoa.

Insinöörintyön työosuuden alkuvaiheessa suunniteltiin WebGL-toteutus, joka tulisi kattamaan jokaisen insinöörityöhön asetetun tavoitteen. Suunnitelma jouduttiin toteamaan liian vaikeaksi tai jopa mahdottomaksi toteuttaa, joten sen sijaan suunniteltiin kaksi eri tavoitteet kattavaa toteutusta, joissa käytettiin hyödyksi siihen mennessä aikaansaatuja komponentteja. Uuden suunnitelman mukaiset natiivi- ja suoratoistoversiot tuottivat myös jonkin verran ongelmia ja ennalta tuntemattoman teknologian käyttöönottamisen haasteita, mutta ne saatiin lopulta kuitenkin onnistuneesti toteutettua uudelleenkäytettäviksi, suurimmalta osin kontitetuiksi moduuleiksi.

Lopputuloksena syntyi sekä reaaliaikaisesti virtuaalista ravintolaa MyCampus-sovellukseen integroitavissa olevan React-käyttöliittymän kautta suoratoistava palvelin että erilliset ladattavissa olevat moninpelimuotoiset interaktiivisuutta tarjoavat sosiaaliset natiivisovellukset. Insinöörityössä käytetyistä suoratoisto- ja konttiteknologioista saatiin runsaasti arvokasta oppia sekä tehdyn tutkimustyön että toteutuksessa esille tulleiden ongelmien ja haasteiden kautta. Työssä selvisi myös suurten 3D-mallien käyttämisen tuomat haasteet, joiden myötä opittiin sekä keinoja 3D-mallien optimoinnille että hyvän suunnittelun tärkeys isoissa projekteissa.

Seuraavaksi, ennen MyCampus-sovelluksen julkaisua, yritetään saada nostettua FFmpeg-Unity-projektin suoratoistettavan kuvan resoluutio Full-HD-tasolle (1920 x 1080 pikseliä) koettamalla poistaa projektissa käytetyn UDP-asiakasohjelman tarve. Myös interaktiivisten natiiviversioiden käyttöliittymiä ja uusia ominaisuuksia kehitetään ennen julkaisua.

Lähteet

About Node.js. Verkkoaineisto. Node.js. <https://nodejs.org/en/about/>. Luettu 26.4.2021.

Docker Subscriptions and Billing FAQs. Verkkoaineisto. Docker. <https://www.docker.com/pricing/faq>. Luettu 3.3.2021.

Enabling Video Streaming for Remote Learning with NGINX and NGINX Plus. Verkkoaineisto. NGINX. <https://www.nginx.com/blog/video-streaming-for-remote-learning-with-nginx/>. Luettu 15.3.2021.

FFmpeg Documentation. Verkkoaineisto. FFmpeg. <https://www.ffmpeg.org/documentation.html>. Luettu 15.3.2021.

Get Started with Docker. Verkkoaineisto. Docker. <https://docs.docker.com/get-started>. Luettu 20.2.2021.

Wiebe, Robert. 2013. Learning Unity 3D. E-kirja. Infinite Skills.

Mirror Documentation. Verkkoaineisto. Mirror. <https://mirror-networking.gitbook.io/docs/>. Luettu 25.1.2021.

Mouat, Adrian. 2015. Using Docker. E-kirja. O'Reilly Media.

NGINX Wiki. Verkkoaineisto. NGINX. <https://www.nginx.com/resources/wiki/ >. Luettu 3.2.2021.

Portnoy, Matthew. 2012. Virtualization Essentials. E-kirja. Sybex.

React Documentation. Verkkoaineisto. React. <https://reactjs.org/docs/>. Luettu 26.4.2021.

Salonen, Ilari. 2020. Tilan virtuaalinen havainnollistaminen lämpökameroiden avulla. Insinöörityö. Metropolia Ammattikorkeakoulu. Theseus-tietokanta.

Unity Manual. Verkkoaineisto. Unity. <https://docs.unity3d.com/Manual/index.html>. Luettu 1.2.2021.

Virtualization. 2019. Verkkoaineisto. IBM Cloud Education. <https://www.ibm.com/cloud/learn/virtualization-a-complete-guide>. 19.6.2019. Luettu 28.2.2021.

What is virtualization. Verkkoaineisto. Red Hat. <https://www.redhat.com/en/topics/virtualization/what-is-virtualization>. Luettu 28.2.2021.