Verkkopelimaailman generointi Unity 2D:llä

Ammattikorkeakoulun opinnäytetyö

Tietojenkäsittelyn koulutusohjelma

Hämeenlinna, kevät 2017

Juho Puoliväli

HÄMEEN AMMATTIKORKEAKOULU
HÄME UNIVERSITY OF APPLIED SCIENCES

Tietojenkäsittelyn koulutusohjelma
Visamäki, Hämeenlinna

Tekijä Juho Puoliväli **Vuosi** 2017

Työn nimi Verkkopelimaailman generointi Unity 2D:llä

Työn ohjaaja Tommi Saksa

TIIVISTELMÄ

Opinnäytetyön tavoitteena oli luoda yksinkertainen verkossa toimiva peli, jonka maailma rakennettaisiin generoimalla manuaalisen rakentamisen sijaan. Työn tarkoitus oli luoda sellainen pelipohja, jota kuka tahansa pystyisi jatkokehittämään omalla tavallaan.

Teoriaosiossa käydään läpi Unity-pelimoottorin ominaisuuksia, kuten käyttöliittymää ja työn kannalta oleellisimpia toimintoja sekä komponentteja, joita se tarjoaa. Teoriaosuudessa käydään läpi myös pelimoottoriin saatavan Photon Unity Networking -työkalun ominaisuuksia ja verkkorajapintaa.

Käytännön osuus on jaettu kahteen osaan. Ensimmäisessä osassa käydään läpi pelimaailman generoiminen koodiesimerkkien ja ruutukaappausten avulla. Toisessa osassa peliin liitetään Photon Unity Networking -työkalu ja muokataan ensimmäisen osion koodit toimimaan rajapinnan mukaisesti.

Unity-pelimoottori ja Photon Unity Networking -työkalu todettiin helppokäyttöiseksi kehitysympäristöksi. Lopputuotoksena on toimiva pelipohja, joka tarjoaa mahdollisuuden jatkokehitykseen.

Avainsanat Unity, Photon Unity Networking, c#, peliohjelmointi, datan generointi

Sivut 35

HAMK
HÄMEEN AMMATTIKORKEAKOULU
HÄME UNIVERSITY OF APPLIED SCIENCES

ABSTRACT

Degree Programme in Business Information Technology
Visamäki, Hämeenlinna

Author Juho Puoliväli **Year** 2017

Subject Multiplayer world generation in Unity2D

Supervisor Tommi Saksa

ABSTRACT

The goal of this thesis was to create a simple online game in which the world would be built by generating instead of manually building it. The purpose was to create template for a game that anyone would be able to further develop in their own way.

The theoretical part goes through Unity game engine features, such as the user interface and thesis relevant functions and components that it offers. The theoretical part also goes through Photon Unity Networking tool features and its network interface.

The practical part is divided into two parts. The first part goes through the world generation with the help of code examples and screenshots. The second part shows integration of the Photon Unity Networking tool and modifications to the first part to make it work with the tools interface.

Unity game engine and Photon Unity Networking tool was found to be easy to use development environment. The end result is a working game template, which provides the opportunity for further development.

Keywords Unity, Photon Unity Networking, c#, game programming, data generation

Pages 35

SISÄLLYS

1 JOHDANTO 1

2 UNITY 2

2.1 Yleistä 2
2.2 Käyttöliittymä ja navigointi 2
2.3 Peliobjekti 6
2.4 Komponentti 6
2.5 Prefab 7

3 PHOTON UNITY NETWORKING 8

3.1 Photon Cloud 8
3.2 Yleistä PUN-ominaisuuksista 8
3.2.1 PhotonView 9
3.2.2 Remote Procedure Call (RPC) 9

4 TOTEUTUS JA TAVOITTEET 10

4.1 Työvälineet ja materiaalit 10
4.2 Työn tavoitteet 11

5 PELIMAAILMAN LUOMINEN 12

5.1 Map-luokka 12
5.2 Block-luokka 14
5.3 WorldGeneration-luokka 15
5.4 WorldDrawer-luokka 20

6 VERKKOON SYNKRONOINTI 24

6.1 PUN-työkalun liittäminen projektiin 24
6.2 Pelaaja peliobjekti 25
6.2.1 Komponentit ja skriptit 25
6.2.2 Liikkuminen 26
6.3 Verkkohuoneet 27
6.4 Verkko-ominaisuuksien lisääminen luokkiin ja peliobjekteihin 29
6.4.1 Peliobjektien luominen ja tuhoaminen verkossa 31

7 YHTEENVETO 33

1 JOHDANTO

Opinnäytetyössä suunniteltiin ja toteutettiin peli Unity 2D -ympäristössä. Aiheen valinnan taustalla oli kiinnostus peliohjelmoinnista, jota olen muutaman vuoden ajan harrastanut. Tutustuin Unity-pelimoottoriin opiskelujeni alkuaikoina, jonka jälkeen aloin vapaa-aikana käyttämään sitä omiin pieniin peliprojekteihini.

Opinnäytetyössä tutustutaan ensin teoriapohjaisesti Unity-pelimoottoriin, sekä siihen liitettävään Photon Unity Networking -työkaluun ja näiden tärkeimpiin toimintoihin ja komponentteihin. Tämän jälkeen tutustutaan algoritmeihin perustuvaan generoimiseen ja lopuksi siihen, miten käytännössä toteutetaan peli näillä työkaluilla 2D-ympäristössä.

Ideana on luoda 2D-pohjainen palikkapeli, kuten esimerkiksi Minecraft ja Terraria. Pelin sisältöä ei rakenneta manuaalisesti, vaan luodaan pelimaailma algoritmien mukaan, jolloin pelin sisältöä voitaisiin luoda automaattisesti, annettujen sääntöjen mukaisesti.

Tämän jälkeen peli alustettaisiin verkkoon käyttäen Photon Unity Networking -työkalua ja verkkoa ylläpidettäisiin Photon Cloud -pilvipalvelussa, joka sisältyy samaan pakettiin. Eli yksi pelaaja rakentaa pelimaailman ja jakaa siitä tietoja muille, eli näin hän isännöi peliä, kun taas muut pelaajat, jotka liittyvät samaan peliin saavat isännältä tiedon generoidusta maailmasta erilaisten synkronointiaskelten kautta.

Työ keskittyy suurimmaksi osaksi itse pelimaailman luontiin ja verkon synkronointiin. Rajaukset tapahtuvat yleisen ulkonäön sekä pelihahmojen kohdalla, koska lopullisen työn tarkoitus on toimia vain lähinnä pelipohjana, jota ulkopuolinen henkilö voisi lähteä kehittämään omilla tavoillaan. Prioriteetit ovat kuitenkin pelimaailman generoinnissa ja sen synkronoinnissa verkkoon. Kaikki tämän jälkeen on jotenkin rajattavissa, kuten pelihahmojen liikkuminen, kontrollit sekä pelin ulkonäkö.

2 UNITY

Unity on Unity Technologiesin kehittämä monialustainen pelimoottori, jolla voidaan kehittää kaksi- ja kolmiulotteisia selain-, konsoli- sekä PC-pelejä. Pelimoottorista on saatavilla ilmainen versio, sekä maksullinen versio nimellä Unity Pro. (Unity 2016a.)

Pelimoottoriin voi myös ladata Asset Storesta ilmaisia ja maksullisia lisäosia, malleja, tekstuureja, skriptejä, animaatioita ja ääniä. (Unity Asset Store 2017.)

2.1 Yleistä

Unity on suuren suosion saavuttanut pelimoottori ja kehitysympäristö, joka pitää sisällään vahvan tuen sekä 3D-grafiikalle että 2D-grafiikalle. Yksi Unityn vahvuuksia on erittäin laaja alustatuki, mikä mahdollistaa pelin kehityksen usealla eri alustalle. Unity on kuitenkin aloittelevalle pelinkehittäjälle helposti lähestyttävä pelimoottori halpojen lisenssien sekä helpon käyttöliittymänsä vuoksi. (Unity 2017b.)

Unityllä voi ohjelmoida kolmella eri ohjelmointikielellä: JavaScript (UnityScript), Boo, ja C#. JavaScript ei ole täysin samanlaista kuin selaimelle tehty JavaScript, Boo muistuttaa Pythonia ja C# muistuttaa paljon javaa, ja tietysti C-kieliä, ja on käytetyin ohjelmointikieli Unityssa. C# pärjää UnityScriptiä paremmin stressitesteissä, ja näin sillä sanotaan olevan parempi suorituskyky vaikka todellisuudessa huomattavaa eroa ei ole. (C# vs Unityscript – Which is faster?)

Työkalujensa puolesta Unity on hyvin kattava pelimoottori. Siitä löytyy sisäänrakennettuna muun muassa törmäyksen tunnistus, fysiikkamoottori, animointityökalu ja valaistuskomponentteja.

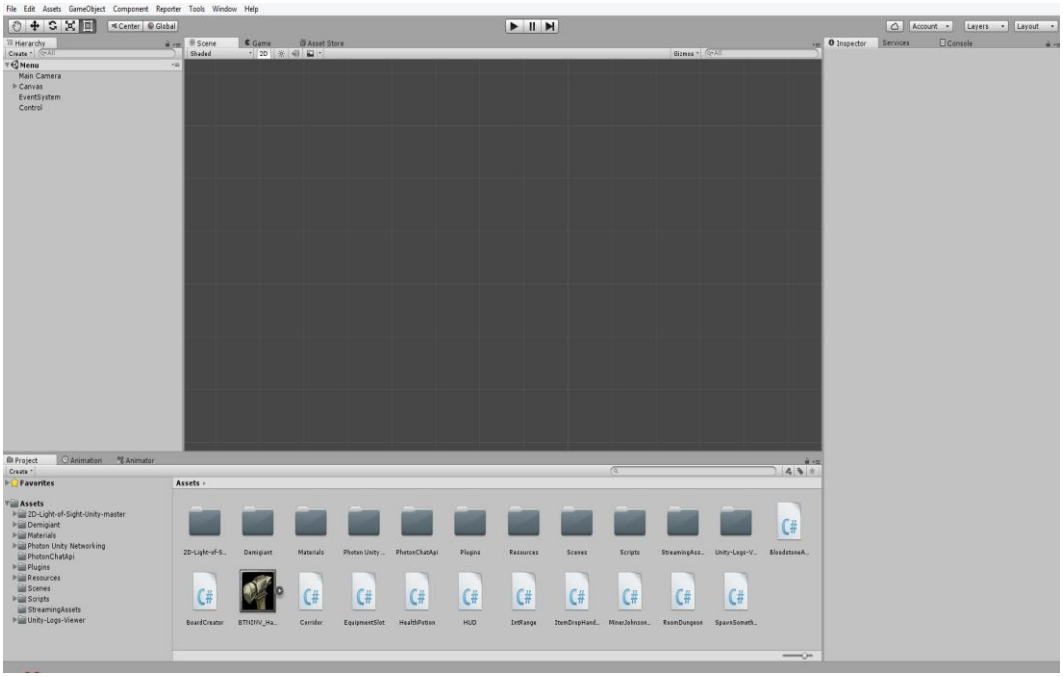
2.2 Käyttöliittymä ja navigointi

Unityn graafisen editorin avulla voidaan muokata kaikkea muuta paitsi pelin koodia. Editorin käyttöliittymää voidaan muokata ja jakaa käyttäjän maun mukaan erikokoisiin ikkunoihin, joiden sijainti sekä välilehdet voidaan myös määrätä.

Kuvassa 1 näkyy esimerkki, miten editorin ikkunat voitaisiin asettaa. Kuvassa editori on jaettu neljään eri pääikkunaan, joissa on omat välilehtensä.

Vasemman reunan ikkunassa näkyy pelin hierarkia välilehti (Hierarchy), jossa näkyvät kaikki kenttään sijoitetut assetit. Sen alapuolella olevassa ikkunassa on välilehti projektin kansioista (Project), josta löytyvät kaikki pelin assetit. Keskellä olevassa ikkunassa on välilehdet pelinäkymästä (Game) ja kenttäikkunasta (Scene). Oikeassa reunassa on valitun peliobjektin tiedot näyttävä Inspector-välilehti. Näillä hallitaan peliobjekteja (GameObject), jotka toimivat rakennuspalikoina kaikille pelin ominaisuuksille. Peliobjekteja ovat mm. esineet, hahmot, valot ja äänet. Peliobjekteista kerrotaan lisää myöhemmässä vaiheessa. Peli rakennetaan kenttiin (Scene), joita voi pitää esillä yksi kerrallaan.

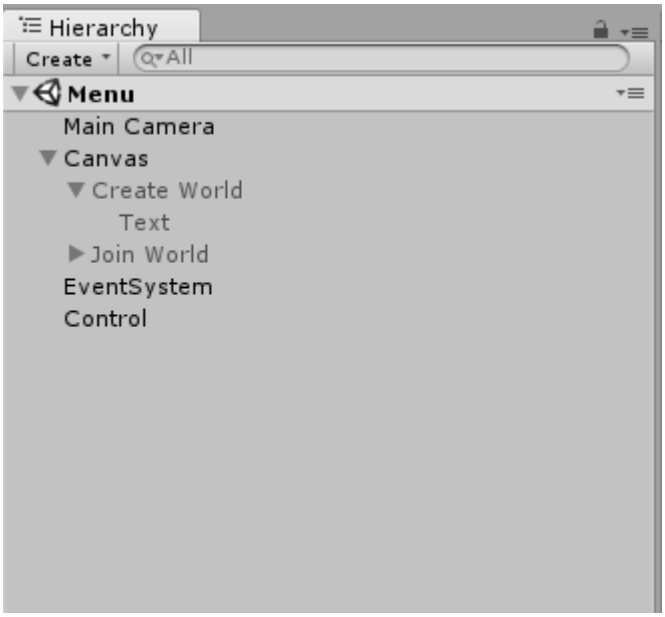
Näiden lisäksi ikkunoiden välilehtiin voidaan maun mukaan lisätä muitakin editorin työkaluja, kuten konsoli sekä animaatiotyökalu.



Kuva 1. Kuvakaappaus Unityn editorinäkymästä.

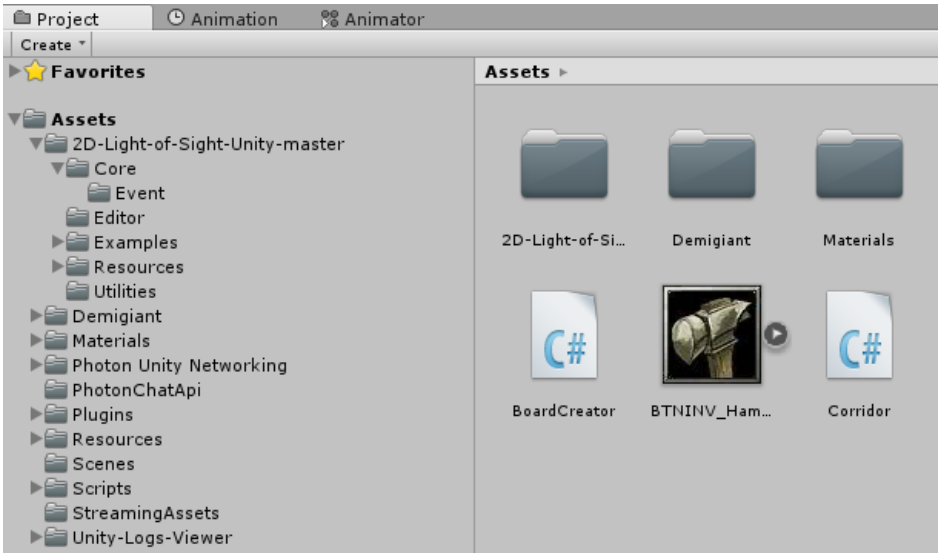
Kun jokin kenttä on valittu, näkyvät kaikki siihen kenttään asetetut objektit Scene- ja Game-näkymissä. Scene-näkymässä on vapaasti ohjattava kamera, jolla voi liikkua kentässä hiiren oikean painikkeen ollessa pohjassa. Game-näkymässä taas ei voi liikkua, sillä sen kameranäkymä on lukittu kentän pääkameraan, joten se siis näyttää aina saman näkymän kuin peliä ajettaisiin.

Hierarchy-näkymässä (Kuva 2) on listattu kaikki kentän peliobjektit. Peliobjektit ovat hierarkisessa järjestyksessä. Yhdellä peliobjektilla voi olla monta aliobjektia eli lasta (Child), mutta korkeintaan yksi vanhempi (Parent). Tämä hierarkia mahdollistaa helpon ja yksinkertaisen peliobjektien ryhmittelyn: kun siirretään vanhempaa peliobjektia, siirtyvät myös sen lapset, rikkomatta rakennetta. (Unity Manual 2016a.)



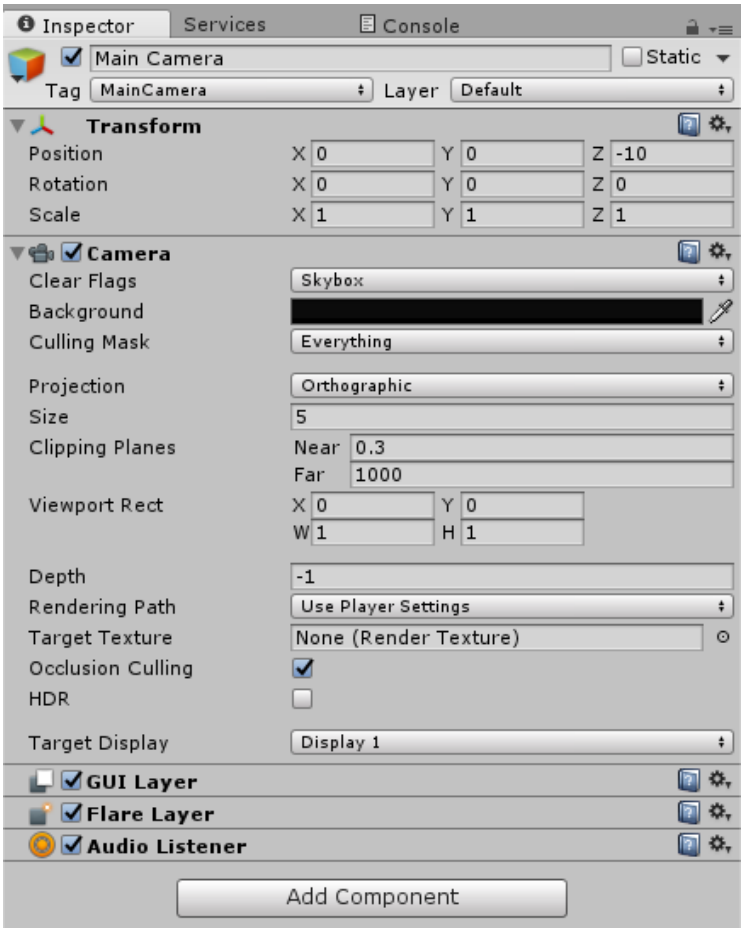
Kuva 2. Kuvakaappaus Hierarchy-näkymästä.

Project-näkymässä (Kuva 3) on listattuna kaikki projektiin kuuluvat kansiot ja tiedostot. Tiedostoja ovat mm. äänet, peliobjektit, skriptit, pelihahmojen sekä esineiden mallit. Tästä näkymästä voi halutessa raahata kenttään peliobjekteja, jolloin raahattu peliobjekti ilmestyy myös projektin Hierarchy- näkymään. (Unity Manual 2016b.)



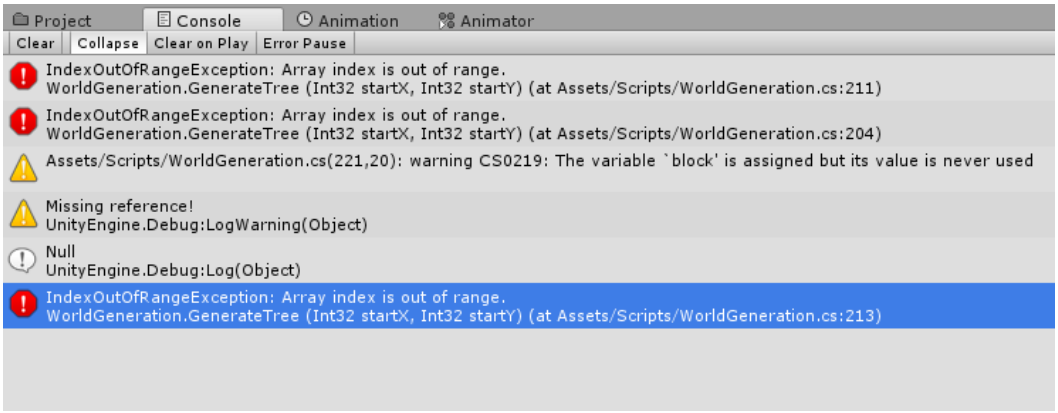
Kuva 3. Kuvakaappaus Project-näkymästä.

Inspector-näkymässä (Kuva 4) näkyvät kaikki valitun objektin tiedot ja komponentit sekä komponenttien tiedot ja arvot. Komponentteja kuvassa ovat Transform, Sprite Renderer, Animator, Circle Collider 2D sekä Rigidbody 2D. Näistä kerrotaan lisää myöhemmässä vaiheessa. (Unity Manual 2016c.)



Kuva 4. Kuvakaappaus Inspector-näkymästä.

Console-näkymässä (Kuva 5) näkyvät virheilmoitukset sekä käyttäjän itse lähettämät viestit ja varoitukset. Se toimii apuna jäljittämään mahdolliset virheet joko skripteistä tai peliobjekteista. (Unity Manual 2016e.)



Kuva 5. Kuvakaappaus Console-näkymästä.

2.3 **Peliobjekti**

Peliobjektit ovat suuri osa Unityä ja kaikki toiminta perustuu niihin. Peliobjekti saadaan luotua editorissa yläpalkista tai suoraan klikkaamalla Hierachyä oikealla hiiren painikkeella ja valitsemalla GameObject → Create Empty. Tällöin kenttään ja Hierarchy-näkymään ilmestyy tyhjä peliobjekti nimellä GameObject. Nyt tätä peliobjektia voi tarkastella Inspector-näkymässä. Tällä hetkellä objektilla on vain aiemmin mainittu pakollinen Transform-komponentti.

Tyhjää peliobjektia voi käyttää esimerkiksi muiden objektien ryhmittelyyn. Tyhjä peliobjekti asetetaan valituille peliobjekteille vanhemmaksi, mikä selkeyttää Hierarchy-näkymän lukemista. Peliobjektia liikuttaessa myös sen lapset liikkuvat.

Tyhjään peliobjektiin voi nyt lisätä käyttäjän haluamia komponentteja Inspector-näkymän Add Component -painikkeella.

2.4 **Komponentti**

Komponentti on yleistermi Unityssä kaikille eri osille, joista peliobjekti koostuu. Ylhäällä olevassa kuvassakin (Kuva 4) näkyy muutamia komponentteja. Kaikista komponenteista tärkein ja ainoa pakollinen on Transform-komponentti. Se kertoo peliobjektin sijainnin suhteessa kentän nollapisteeseen tai vanhempaan, jos sellainen on olemassa. Transform-komponentti myös määrittää peliobjektin asennon ja koon.

Muita tärkeitä komponentteja ovat Rigidbody, joka kertoo fysiikkamoottorille tarpeellista tietoa peliobjektista, kuten massan ja voiman, esimerkiksi törmäyksiä varten. Rigidbodyyn liittyvä komponentti, Collider, kertoo tarkemmin, mitä törmäyksen jälkeen tulee tapahtua.

Myös erilaiset valo- ja partikkeliefektejä tuottavat osat liitetään peliobjekteihin komponentteina. Myös pelinäkymän kamera on komponentti. Peliobjekteihin liitetään myös scriptejä komponentteina. (Unity Manual 2016d.)

2.5 **Prefab**

Kun peliobjekteja rupeaa kertymään peliin enemmän ja enemmän, ja varsinkin silloin kun samanlaisia peliobjekteja luodaan, kuten esimerkiksi puut, kivet ja pensaat, rupeaa kehitys olemaan työlästä ja sekavaa. Tätä varten Unity tarjoaa toimintoa Prefab.

Prefab mahdollistaa peliobjektin säilömisen, siten että se säilyttää kaikki siihen liitetyt komponentit ja ominaisuudet. Prefab siis tallentaa tehdyn peliobjektin ja muuntaa sen Prefab muotoon pelin tiedostoihin, josta voit myöhemmin hakea sitä ja uudelleen käyttää. Pelin tiedostoissa voi olla esimerkiksi puu-Prefab. Hiirellä raahaamalla pystyt tekemään identtisiä kopioita tästä puusta peliin, samalla kun säilytät puun pohja Prefabin tiedostoissa. Kun muokkaa tätä pohjaa, kaikki pelissä olevat puut jotka on luotu tätä pohjaa käyttäen päivittyvät. Mutta tämä ei tarkoita sitä, että kaikki puut olisivat aina samanlaisia. Prefabistä tehtyjä kopioita voi muokata myös yksittäin Scene-näkymässä, muokkaamatta toisia. Jos pohja-Prefabia muokkaa, niin kaikki siitä luodut peliobjektit muokkautuvat myös.

3 PHOTON UNITY NETWORKING

Photon Unity Networking (PUN) on Unity-pelimoottoriin saatava työkalu, joka korvaa ja parantaa Unityn omia verkkotoimintoja. Paketin ominaisuuksiin kuuluu lähinnä kommunikointi ja pelaajien yhdistäminen samoihin verkkoihin. Paketin ohjelmointirajapinta on läheinen Unityn omaan verkkorajapintaan, joka mahdollistaa helpon käyttöönoton. Paketti tarjoaa myös automaattista muuntajaa auttamaan vanhojen Unity projektien siirron paketin tarjoamaan verkkoympäristöön. Paketti on ladattavissa aiemmin mainitusta Unity Asset Storesta, ja siitä on saatavilla sekä ilmainen PUN FREE, sekä maksullinen PUN PLUS -versio. Maksullinen versio tarjoaa tukea suuremmalle pelaajamäärälle verkossa.
(Photon Unity Networking 2017a.)

3.1 Photon Cloud

Kaikki projektit jota työstät Photon Unity Networking -paketilla, yhdistyvät tämän paketin mukana tulevaan Photon Cloud -pilvipalveluun. Tämä palvelu hoitaa kaiken tarvittavan projektin verkkoisännöinnistä, operaatioista sekä skaalautumisesta, jolloin käyttäjälle jää enemmän aikaa keskittyä pelin luomiseen. (Photon Unity Networking 2017b.)

Kaikki Photon Cloud -pilvipalvelulla tehdyt projektit pohjautuvat asiakas-palvelinarkkitehtuuriin. Vertaisverkkoarkkitehtuureissa asiakkaat eivät usein pysty yhdistymään osoitteenmuunnosongelmien vuoksi. Tämä ongelma on vielä pahempi matkapuhelin verkoissa, mutta Photon Cloud -palvelu korjaa nämä ongelmat. Palvelun toimintaa myös seurataan ja ylläpidetään jatkuvasti, mikä parantaa kokemusta merkittävästi. (Photon Unity Networking 2017c.)

3.2 Yleistä PUN-ominaisuuksista

PUN tarjoaa kattavan kokoelman luokkia, metodeja ja jopa valmiita malliskriptejä, kaikkeen mahdolliseen toimintaan verkossa. Mutta kerron tarkemmin vain oleellisimmista ominaisuuksista ja komponenteista joita PUN tarjoaa.

3.2.1 PhotonView

PhotonView on yksi monista PUN-työkalun tarjoamista komponenteista joita voi liittää peliobjekteihin. Kun peliobjektiin liittää tämän komponentin, se saa ominaisuuden kommunikoida muiden tämän komponentin omaavien peliobjektien kanssa.

PhotonView on skripti komponentti, jota käytetään viestien lähettämiseen. Se täytyy liittää peliobjektiin tai Prefabiin. Se on hyvin läheinen Unityn NetworkView -komponentin kanssa. Ainakin yksi PhotonView-komponentti pitää olla läsnä, jotta ajonaikainen viestintä tai vaihtoehtoisesti uusien PhotonView-komponenttien luominen ja jakaminen onnistuisivat. (Photon Unity Networking 2017d.)

3.2.2 Remote Procedure Call (RPC)

Remote Procedure Calls (RPC) mahdollistaa verkossa olevien peliobjektien metodien kutsumisen etäältä. Tämä on hyödyllistä metodeissa joita ei kutsuta liian usein ja joista pitää lähettää kutsu myös muille verkossa oleville. Hyvänä esimerkkinä on pelaajan hyppiminen ja ampuminen koska nämä toiminnot perustuvat yleensä käyttäjän manuaalisesti tehtyihin toimintoihin, kuten hiiren tai näppäimistön painallukseen. Muut pelaajat verkossa eivät tiedä koska kukin pelaaja painaa mitäkin näppäintä ilman että siitä erikseen kerrotaan.
Ratkaisu tähän on liittää pelaajan painallukset RPC-metodiin, jolloin hiirtä painaessa jokaisen pelaajan kohdalla suoritetaan tämä RPC. Tätä toimintoa ei tulisi kuitenkaan käyttää toiminnoissa joita tapahtuu usein, koska tiiviisti lähetetyt RPC-paketit kuormittavat verkkoa.

RPC-paketeilla on kaikilla sama yksinkertainen tavoite: tehdä koodin suorittamisesta etäisellä koneella mahdollisimman yksinkertaista ja helppoa, aivan kuin kutsuisi metodia lokaalisti. RPC kutsuu ainoastaan metodeja skripteissä, jotka ovat liitettynä samaan peliobjektiin kuin PhotonView.

4 TOTEUTUS JA TAVOITTEET

Opinnäytetyön käytännön toteutus on kaksivaiheinen. Ensimmäisessä vaiheessa suunnittelen ja kehitän skriptejä tukemaan pelimaailman luomisen eri vaiheita. Tämä vaihe on hyvin ohjelmointipohjaista eikä pelin verkkopuolen toimintoihin ja synkronointiin perehdytä vielä ollenkaan, joten tavoitteen lopullinen toteutus tapahtuu vasta toisessa vaiheessa.

Opinnäytetyön toisessa vaiheessa lisään työhön verkkotyökalun, ja sen tarjoamilla toiminnoilla ja komponenteilla muokkaan ensimmäisessä vaiheessa luotuja skriptejä ja peliobjekteja, niin että ne toimivat myös verkkoympäristössä.

4.1 Työvälineet ja materiaalit

Teen pelin suurimmalta osin Unity-pelimoottorilla ja siitä saatavilla olevalla versiolla 5.4.2f2. Kaikki peliin tarvittavat skriptit teen Microsoft Visual Studio 2015 -ohjelmankehitysympäristössä. Unity tukee täysin näiden ympäristöjen yhteistyötä, joten ongelmia skriptien yhteensopivuudessa ei pitäisi tapahtua.

Pelin verkkoon alustamisen, synkronoinnin ja kaikki verkkopohjaiset toiminnat tehdään Photon Unity Networking -työkalulla ja sen mukana tulevalla Photon Cloud -pilvipalvelulla.

Aiemmin mainittujen työkalujen ja ohjelmien lisäksi tarvitsen peliä varten tekstuurit peliobjekteille, jotka aion luoda Gimp -ohjelmalla. Tarvittaviin materiaaleihin kuuluvat ainoastaan peliobjektien tekstuurit, joiden ei työn kannalta tarvitse olla kovinkaan hienoja, koska opinnäytetyö keskittyy enemmän pelin toimintaan kuin ulkonäöllisiin yksityiskohtiin.

Työn vaatimiin tekstuureihin kuuluvat ruohon, maaperän ja puiden tekstuurit. Näistä teen hyvin yksinkertaisia neliön muotoisia kuvia siten, että ruoho olisi vihreän, maaperä harmaan ja puu ruskean värinen.

4.2 Työn tavoitteet

Ensimmäisen vaiheen tavoitteena tulisi saada tehtyä ainakin seuraavat skriptit tukemaan maailman luomisen kaikkia eri vaiheita. Skriptit on nimetty toistaiseksi niiden tarkoituksen mukaan, ja saatan muuttaa niitä kesken työn, jos siihen on tarvetta.

Map-skripti sisältäisi ainakin pituuden, leveyden ja kaksiulotteisen taulukon. Luokasta tehtäisiin pelimaailman luomisprosessin alussa uusi olio ja sille annettaisiin pituus ja leveys kokonaislukuina, jonka jälkeen se täydentäisi annettujen pituuden ja leveyden mukaan sisältämänsä taulukon nollilla. Tämän luokan tarkoituksena olisi säilöä tieto siitä missä on mikäkin palikka pelimaailmassa.

WorldGeneration-skripti olisi vastuussa aiemmin mainitun Map-luokan taulukon täyttämisestä. Luokka muodostuisi vaiheistetusta datan luomisesta, käyttäen erilaisia algoritmejä apuna. Luokalle annettaisiin tietoa, kuten kuinka paljon luolia ja puita pelimaailmassa pitäisi olla, ja muuttaisi prosessiaan niiden mukaan. Annettujen tietojen ja algoritmien perusteella luokka päättelisi minkä arvon se asettaa Map-luokan taulukkoon mihinkin kohtaan. Esimerkiksi, jos määritellään että numero 1 olisi ruohoa, ja luokka asettaisi tämän arvon taulukon ensimmäiselle riville ja ensimmäiselle sarakkeelle, tarkoittaisi että pelimaailman vasen yläkulma tulisi olla ruohoa. Tällä samalla periaatteella se täyttäisi Map-luokan taulukon täyteen erilaisia arvoja nollasta ylöspäin, viitaten mitä palikkaa missäkin osassa pelimaailmaa sijaitsee, vai onko sijainti tyhjä. WorldGeneration-skriptin tekemiseen tulen käyttämään apuna Unityn kotisivulta löytyvää tutoriaalia (Cellular Automata 2016).

WorldDrawer-skripti olisi vastuussa pelin prefabien säilömisestä ja näiden asettelusta pelimaailmaan. Luokka käyttäisi apunaan valmiiksi täydennettyä Map-luokan taulukkoa ja tarkistaisi yksi kohta kerrallaan, että mitä prefabia pitäisi asettaa mihinkin kohtaan pelimaailmaa.

Jos nämä vaiheet onnistuisivat niin kuin niiden pitäisi, olisi tuloksena koodipohjaisesti generoitu pelimaailma, joka koostuisi erivärisistä palikoista.
Palikat sijaitsisivat toistensa päällä ja vieressä, ruudukkomaisesti. Myös tyhjiä kohtia pelimaailmasta löytyisi, ja nämä tyhjät kohdat muodostaisivat luolia ja kukkuloita pelimaailmaan.

Toisen vaiheen tavoitteena on lisätä peliin verkko-ominaisuudet onnistuneesti. Näihin ominaisuuksiin kuuluisivat aiemmin mainitun Map-luokan tietojen jako pelaajien kesken, jotta pelaajat osaisivat rakentaa pelimaailman samankaltaisena kuin se on muilla. Tämän lisäksi pelaajien toiminnat, kuten palikoiden tuhoaminen ja asettelu pelimaailmassa, pitäisi jakaantua kaikkien pelaajien kesken, jotta pelimaailma pysyisi aina synkronoituna.

5 PELIMAAILMAN LUOMINEN

Tapa millä lähdin kehittämään pelimaailman luomista, oli generoida arvoja kaksiulotteiseen taulukkoon. Ensin peli asettaa taulukkoon numeraalisia arvoja algoritmien mukaan viittaamaan, mikä peliobjekti on kyseessä. Taulukon arvo yksi vastaa pelimaailmassa kiveä ja arvo kaksi vastaa pelimaailmassa ruohoa. Taulukon sisältämiä arvoja käyttää toinen skripti hyödykseen lukemalla ne läpi ja luomalla pelimaailman niiden mukaisesti. Arvojen sijainti taulukossa määrittää vastaavan sijainnin pelimaailmassa.

Koska teen työn kokonaan 2D-ympäristössä, oli tapa mielestäni loogisin ja helpoiten lähestyttävä, joten muita tapoja ei tarvinnut alkaa edes harkitsemaan. Tavoitteena oli myös se, että pelimaailma koostuisi neliön muotoisista peliobjekteista, joten kaksiulotteinen taulukko viittaisi suoraan missä mikäkin peliobjekti sijaitsee pelimaailmassa.

5.1 Map-luokka

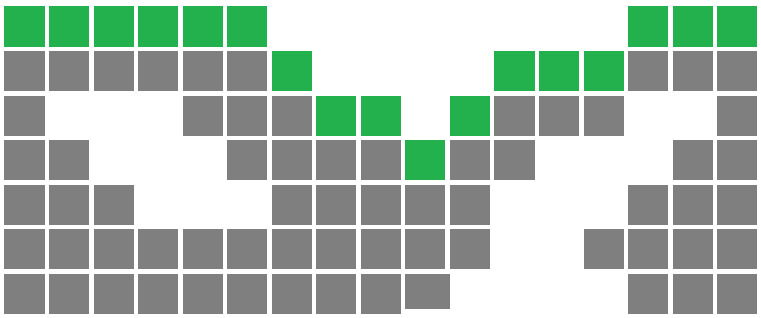
Onnistuakseni jakamaan pelimaailman pelaajien kesken, tarvitsin tavan säilöä sen tietoja yksinkertaisessa muodossa ilman että se olisi liian raskasta verkolle. Vaikka tässä vaiheessa en vielä perehtynyt verkko-ominaisuuksiin, oli se silti hyvä pitää mielessä.

Päädyin ratkaisuun jossa tein luokan nimellä Map (Koodiesimerkki 1), joka pitää sisällään kaksiulotteista blocks-numerotaulukkoa. Taulukko säilöö tiedon kaikista pelimaailman muodostavista palikoista. Taulukon arvo viittaa peliobjektiin, kun taas arvon sijainti taulukossa viittaa peliobjektin sijaintiin pelimaailmassa (Kuva 6 ja kuva 7). Sijainti ei ole täysin tarkka, koska taulukon rivit ja sarakkeet ovat kokonaislukuja nollasta ylöspäin. Tämän vuoksi ylläpidin sääntöä, jossa lasken taulukon rivin ja sarakkeen vastaavasti palikan leveydellä ja pituudella, jolloin palikat olivat aina kiinni toisissaan mutta ei koskaan päällekkäin tai irti toisistaan. Kyseistä sääntöä ei toteuteta tässä luokassa, koska luokka ainoastaan säilöö tietoa, jättäen tiedon käyttö muille luokille.

```
int width;
int height;
int[,] blocks;
int[,] checkMap;

public Map(int width, int height)
{
    Width = width;
    Height = height;
    Blocks = new int[width, height];
    CheckMap = new int[width, height];
}
```

Koodiesimerkki 1. Map-luokan muuttujat ja konstruktori.



Kuva 6. Esimerkki miten palikat sijoittuvat pelimaailmassa.

```
1 1 1 1 1 1 0 0 0 0 0 0 0 0 1 1 1
2 2 2 2 2 2 1 0 0 0 0 1 1 1 2 2 2
2 0 0 0 2 2 2 1 1 0 1 2 2 2 0 0 2
2 2 0 0 0 2 2 2 2 1 2 2 0 0 0 2 2
2 2 2 0 0 0 2 2 2 2 2 0 0 0 2 2 2
2 2 2 2 2 2 2 2 2 2 2 0 0 2 2 2 2
2 2 2 2 2 2 2 2 2 2 0 0 0 0 2 2 2
```

Kuva 7. Esimerkki miten palikoiden arvot sijoittuvat taulukossa.

Kaksiulotteisen blocks-taulukon lisäksi, Map-luokka pitää sisällään width- ja height-muuttujat, sekä toisen kaksiulotteisen checkMap-taulukon (Koodiesimerkki 1). Width-muuttuja määrittää pelimaailman leveyden ja height-muuttuja pelimaailman korkeuden. CheckMap-taulukko on samankaltainen kuin blocks-taulukko, mutta se pitää sisällään vain arvoja nolla ja yksi. Taulukko on asetettu ensin täyteen nollilla, mutta pelimaailman generoinnin aikana sen arvot muuttuvat ykkösiksi, vastaamaan sitä, että kyseinen kohta on generoitu. Taulukon avulla peli tietää jättää vastaavat kohdat generoimatta uudestaan, joka estää samojen palikoiden jatkuvan luomisen päällekkäin.

5.2 **Block-luokka**

Opinnäytetyön tarkoituksena oli luoda palikoista koostuva pelimaailma. Palikat sijoittuisivat pelimaailmassa ruudukon tavoin, toistensa päällä, alla tai vieressä. Työ tarvitsi peliobjekteja jotka olisivat palikan muotoisia ja erivärisiä, vastaamaan mitä materiaalia ne olisivat. Ruoho olisi vihreä palikka ja maaperä harmaa palikka. Koska pelimaailma tulisi koostumaan pelkästään samankokoisista palikoista ja ne omistaisivat lähes samat ominaisuudet, tarvitsin vain yhden ”pää-peliobjektin” josta pystyisin luomaan muita peliobjekteja pelkästään kuvaa ja väriä vaihtamalla.

Lähdin siis tekemään uutta peliobjektia johon lisäsin aluksi Sprite Renderer -komponentin. Tämä komponentti antaa peliobjektille visuaalisen ulkonäön perustuen jonkinlaiseen kuvaan tai muuhun graafiseen lähteeseen. Tein yksinkertaisen neliön muotoisen kuvan Gimp-kuvankäsittelyohjelmalla. Lisäsin kuvan Unity-projektiin PNG-muodossa ja annoin sen peliobjektin ulkonääksi.

Peliobjekti tarvitsi myös ominaisuuden joka antaisi pelaajille mahdollisuuden kävellä sen päällä ja etteivät pelaajat kävelisi sen läpi, joten lisäsin peliobjektiin Box Collider2D -komponentin mahdollistamaan törmäykset muihin peliobjekteihin, kunhan ne myöskin omistavat kyseisen komponentin. Komponenttia lisätessä Unity muokkaa törmäyksen rajat automaattisesti perustuen peliobjektin graafiseen lähteeseen, joka on tässä tapauksessa Sprite Renderin -kuva.

Peliobjektin ulkonäön luomisen ja törmäyksen tunnistuksen jälkeen se tarvitsi skriptin tukemaan koodipohjaisia toimintoja. Lisäsin peliobjektiin uuden skriptin nimellä Block (Koodiesimerkki 2). Tähän luokkaan asetin kaksi muuttujaa nimellä Width ja Height, joiden arvot viittaavat peliobjektin kokoon pelimaailmassa. Kokoa tullaan käyttämään peliobjekteja asetettaessa pelimaailmaan. Luokka sisältää myös OnMouseDown-metodin jota Unity kutsuu automaattisesti peliobjektia painettaessa hiirellä. Tässä tapauksessa metodia kutsuttaessa, peliobjekti tuhoutuu.

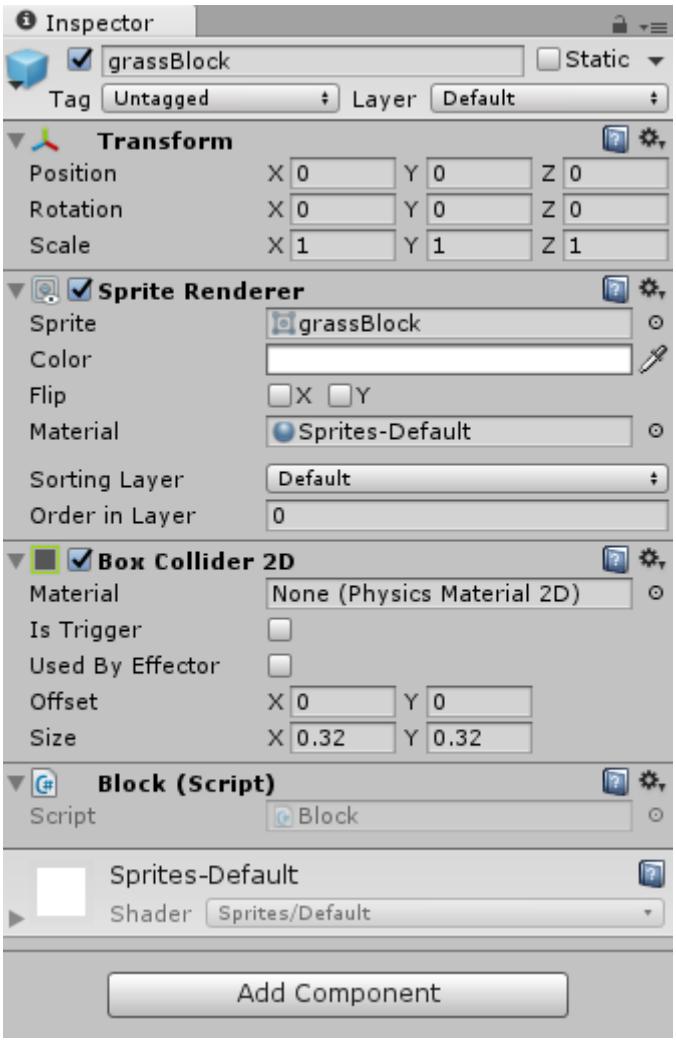
Peliobjektin tultua valmiiksi raahasin sen Scene-näkymästä projektin tiedostoihin, jolloin peliobjektista syntyi prefab (Kuva 8). Seuraavaksi kopioin prefabin ja muokkasin nyt omistamani kaksi prefabiä niiden tarkoituksen mukaan, eli annoin niille oman nimen ja värin. Tämän jälkeen oli pelin kannalta oleellisimmat peliobjektit valmiit.

```
public class Block : MonoBehaviour {

    public static readonly float Width = 0.32f;
    public static readonly float Height = 0.32f;

    void OnMouseDown()
    {
        WorldDrawer.Instance.DestroyBlock(this.gameObject);
    }
}
```

Koodiesimerkki 2. Block-luokka.



Kuva 8. Kuvakaappaus valmiista Ruoho-prefabistä.

5.3 WorldGeneration-luokka

Seuraava vaihe oli luoda pelin kannalta suurin ja tärkein luokka, joka generoisi tarvittavat tiedot pelimaailman luomista varten. Tämä luokka keskittyy pelkästään täyttämään pelimaailman tietoja aiemmin mainittuun Map-luokkaan. Itse WorldGeneration-luokka ei käytä näitä tietoja enempää, koska sen tarkoitus on vain luoda dataa.

Luokka sisältää erilaisia muuttujia joilla saadaan pelimaailma muodostumaan halutun mukaiseksi. Luokka sisältää myös Start-metodin jonka pelimoottori suorittaa heti peliä ajettaessa. Kuten koodiesimerkistä 3 näkyy, on luokalle ensin annettu staattinen muuttuja nimellä Instance johon asetetaan viittaus tähän luokkaan Start-metodissa. Tämä mahdollistaa helpon yhteyden muista luokista ilman olioiden luomista, joka tässä tapauksessa jopa haittaisi pelin toimintaa. Tämän takia Instance-muuttujan jälkeen on tyhjä konstruktori estämässä olioiden tekemisen tästä luokasta koska tarkoituksena oli että peli sisältäisi vain yhden WorldGeneration-luokan.

Luokka sisältää myös muuttujat leveydelle ja korkeudelle; worldWidth ja worldHeight. Nämä muuttujat määrittävät kuinka suuri Map-luokan taulukko tulee olemaan ja näin myös määrittää itse pelimaailman suuruuden.
Seuraavaksi luokka sisältää string-muuttujan seed, sekä boolean-muuttujan useRandomSeed. Seed-muuttujaa käytetään generoinnissa apuna kun halutaan tehdä jotain näennäissatunnaista. Sitä käytetään mukana generoinnin algoritmissä siten että sen arvo vastaa aina tiettyä generoinnin lopputulosta, jolloin samaa arvoa käytettäessä saadaan myös samanlainen generoinnin lopputulos. Luokan useRandomSeed-muuttujan arvo määrää halutaanko käyttää manuaalisesti annettavaa seed-arvoa vai annettaanko pelin itse arpoa se.

Luokan randomFillPercent-muuttuja määrittää kuinka paljon pelimaailma tulee sisältämään luolia ja muita muodostelmia jotka ovat pelimaailmassa tyhjiä. Muuttuja on numeraalinen ja se on rajattu nollan ja sadan välille. Mitä pienempi muuttujan arvo on, sitä vähemmän luokka tulee täyttämään Map-luokan taulukkoa nollalla jolloin pelimaailma tulee olemaan tyhjempi.

Luokan map-muuttuja säilöö viittauksen Map-luokan olioon joka luodaan Start-metodissa antaen Map-luokan konstruktorille aiemmin mainitut leveys sekä korkeus muuttujat. Tämä Map-luokan olio säilöö kaiken tiedon pelimaailmasta, ja tätä samaa oliota käytetään aina kun luodaan tai muokataan pelimaailmaa.

```
public static WorldGeneration Instance;
private WorldGeneration() { }

[SerializeField]
private int worldWidth;
[SerializeField]
private int worldHeight;

public string seed;
public bool useRandomSeed;

[Range(0, 100)]
public int randomFillPercent;

public Map map;

void Awake()
{
    Instance = this;
    map = new Map(WorldWidth, WorldHeight);
}
```

Koodiesimerkki 3: WorldGeneration-luokan muuttujat ja Start-metodi.

Awake-metodin (Koodiesimerkki 3) lopussa luokka kutsuu GenerateMap-metodia (Koodiesimerkki 4). Tämä metodi on kolmivaiheinen; ensin se täyttää Map-luokan taulukon alustavasti kutsumalla RandomFillMap-metodia, jonka jälkeen se tasoittaa taulukon arvoja luonnollisemmin kutsumalla SmoothMap-metodia viisi kertaa ja lopuksi se viimeistelee taulukon yksityiskohdilla kutsumalla GenerateGrass-metodia.

```
void GenerateMap()
{
    RandomFillMap();
    for (int i = 0; i < 5; i++)
    {
        SmoothMap();
    }
    GenerateGrass();
}
```

Koodiesimerkki 4. GenerateMap-metodi.

RandomFillMap-metodi (Koodiesimerkki 5) tarkistaa ensin käytetäänkö manuaalista seed-muuttujaa vai täytyykö sille luoda uusi arvo. Tällä seed-arvolla luodaan uusi System.Random-olio jolla pystytään suorittamaan näennäissatunnaisia toimintoja joita tämä metodi tarvitsee. Metodi iteroi map-olion taulukon läpi ja asettaa sinne arvon nolla tai yksi, perustuen aiemmin mainittuun randomFillPercent-muuttujan arvoon ja System.Random-olioon. Lopputuloksena map-olion taulukko saa alustavan pohjan arvoja yhden ja nollan väliltä.

```
void RandomFillMap()
{
    if (useRandomSeed)
    {
        seed = System.DateTime.Now.Millisecond.ToString();
    }

    System.Random pseudoRandom =
            new System.Random(seed.GetHashCode());

    for (int x = 0; x < WorldWidth; x++)
    {
        for (int y = 0; y < WorldHeight; y++)
        {
            if (x == 0 || x == WorldWidth - 1 || y == 0
                        || y >= WorldHeight - 15)
            {
                if (y == WorldHeight - 5)
                {
                    map.Blocks[x, y] = 0;
                }
                else
                {
                    map.Blocks[x, y] = 0;
                }
            }
            else
            {
                map.Blocks[x, y] =
                    (pseudoRandom.Next(0, 100) <
                         randomFillPercent) ? 1 : 0;
            }
        }
    }
}
```

Koodiesimerkki 5. RandomFillMap-metodi.

SmoothMap-metodi (Koodiesimerkki 6) käy map-olion alustetussa vaiheessa olevan taulukon uudelleen läpi. Metodin tarkoituksena on tasoittaa taulukon arvoja siten että ne vaikuttaisivat olevan yhteydessä toisiinsa. Jos jokin taulukon arvoista on nolla, olisi sen lähellä suurempi mahdollisuus olla nollia kuin ykkösiä. Tämä sama tapa toimisi myös toisinpäin, eli ykkösien läheisyydessä esiintyisi enemmän ykkösiä. Metodin lopullisena tuloksena taulukon arvot muodostaisivat luonnolliselta vaikuttavia muodostelmia. Työtä tehdessä huomasin että sain parempia tuloksia generoinnista kun kutsuin tätä metodia useammin.

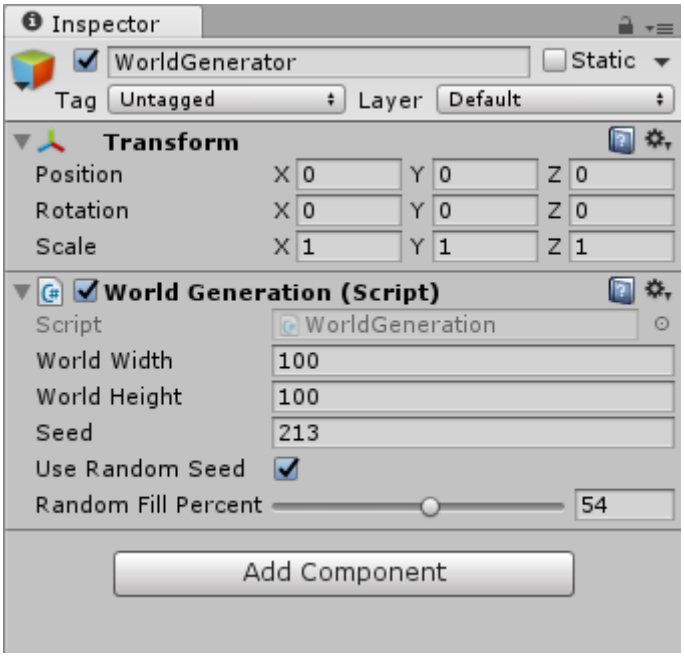
```
void SmoothMap()
{
    for (int x = 0; x < WorldWidth; x++)
    {
        for (int y = 0; y < WorldHeight - 15; y++)
        {
            int neighbourWallTiles = GetSurroundingWallCount(x, y);
            if (neighbourWallTiles > 4)
                map.Blocks[x, y] = 1;
            else if (neighbourWallTiles < 4)
                map.Blocks[x, y] = 0;
        }
    }
}
```

Koodiesimerkki 6. SmoothMap-metodi.

Tuloksen saavuttamiseksi metodi käyttää apunaan GetSurroundingWall-Count-metodia (Koodiesimerkki 7) joka laskee kuinka monta ykköstä kunkin map-olion taulukon arvon ympärillä on. Jos ykkösiä löytyy enemmän kuin neljä, metodi muuttaa kyseisen taulukon arvon ykköseksi. Jos taas ykkösiä löytyy vähemmän kuin neljä, muuttaa metodi kyseisen arvon nollaksi.

```
int GetSurroundingWallCount(int gridX, int gridY)
{
    int wallCount = 0;
    for (int neighbourX = gridX - 1; neighbourX <= gridX + 1; neigh-
bourX++)
    {
        for (int neighbourY = gridY - 1; neighbourY <= gridY + 1;
neighbourY++)
        {
            if (neighbourX >= 0 && neighbourX
                < WorldWidth && neighbourY
                >= 0 && neighbourY
                < WorldHeight)
            {
                if (neighbourX != gridX || neighbourY != gridY)
                {
                    wallCount += map.Blocks[neighbourX, neighbourY];
                }
            }
            else
            {
                wallCount++;
            }
        }
    }
    return wallCount;
}
```

Koodiesimerkki 7. GetSurroundingWallCount-metodi.



Kuva 9. WorldGeneration-skripti peliobjektiin liitettynä.

5.4 WorldDrawer-luokka

Kun sain pelimaailman tiedot tallentumaan onnistuneesti aiemmin mainittuun Map-taulukkoon, oli seuraavaksi vuorossa itse pelimaailman luominen tämän kyseisen taulukon pohjalta. Vielä ei siis ollut minkäänlaista visuaalista tulosta pelissä, vaan ainoastaan dataa taustalla. Vuorossa oli siis pelimaailman ”piirtäminen” taulukon arvojen mukaan. Jokainen taulukon arvo piti saada viittaamaan tietynlaiseen peliobjektiin pelimaailmassa. Päätin tehdä uuden luokan nimellä WorldDrawer, jonka tehtävänä olisi pelkästään pelin prefabien säilöminen ja asettelu pelimaailmaan Map-taulukon arvojen perusteella.

WorldDrawer-luokka (Koodiesimerkki 8) on hyvin yksinkertainen ja se ei sisällä paljoa muuttujia tai viittauksia muihin luokkiin, vaan ainoastaan muutamia toimintoja hallinnoimaan kaikkia pelin visuaalisia elementtejä. Oleellisena luokan ominaisuutena on kaikkien pelin prefabien säilöminen listassa. Luokka säilöö siis kaikkia ”rakennuspalikoita” joita peli tarvitsee ja myöskin asettaa ja tuhoaa niitä tarpeiden mukaan eri metodeilla.

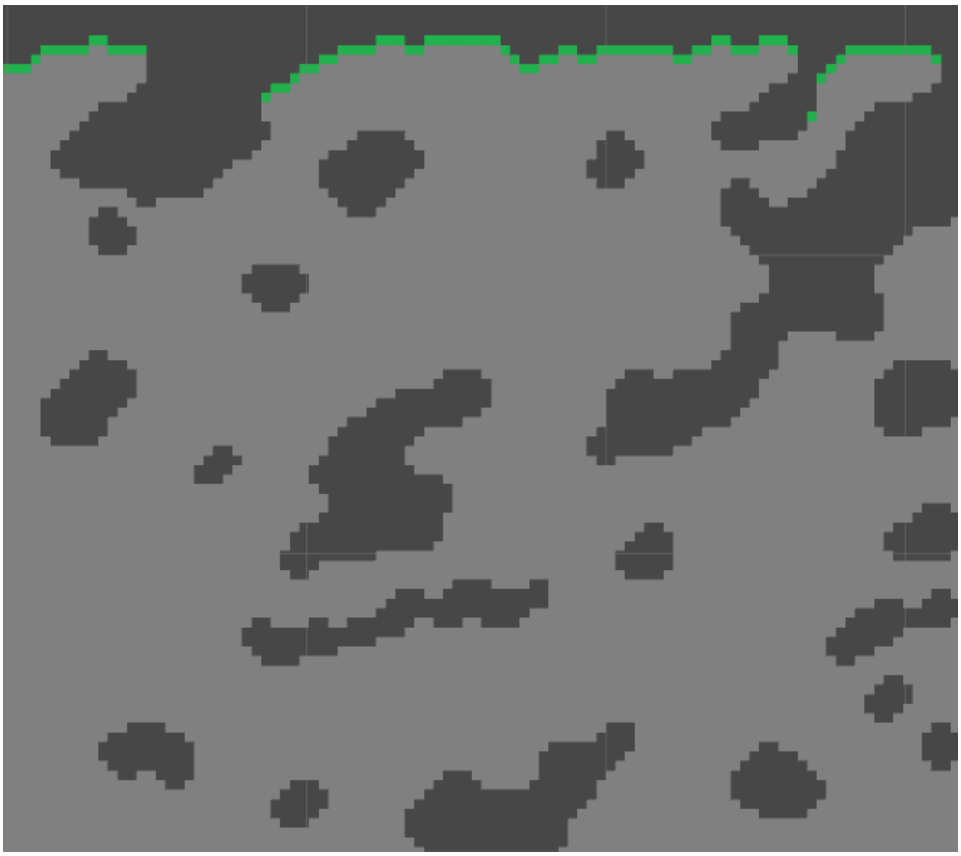
```
public static WorldDrawer Instance;
private WorldDrawer() { }

Map map;
public List<GameObject> blockPrefabs = new List<GameObject>();

void Awake()
{
    Instance = this;
}
```

Koodiesimerkki 8: WorldDrawer-luokan muuttujat ja Awake-metodi.

DrawWorld-metodin (Koodiesimerkki 9) tehtävänä on ”piirtää” pelimaailman tämän hetkinen tila perustuen parametrina annettuun Map-luokkaan, joka toimii pelin ”karttana”. Metodia kutsutaan ainoastaan peliä käynnistäessä, joten jälkeenpäin tehdyt muutokset pelimaailmaan tai karttaan ei päivity. Pelin käynnistyessä muodostuu pelimaailma kartan mukaisesti (Kuva 10).



Kuva 10. Kuvakaappaus pelimaailmasta.

```
public void DrawWorld(Map _map)
{
    this.map = _map;
    if (map == null) { return; }

    for (int x = 0; x < map.Width; x++)
    {
        for (int y = 0; y < map.Height; y++)
        {
            if (map.Blocks[x, y] !=
                0 && map.CheckMap[x, y] == 0)
            {
                Vector2 blockPos = new Vector2(
                    x * Block.Width,
                    y * Block.Height);

                map.CheckMap[x, y] = 1;
                SpawnBlock(blockPos, map.Blocks[x, y]);
            }
        }
    }
}
```

Koodiesimerkki 9: DrawWorld-metodi.

SpawnBlock-metodin (Koodiesimerkki 10) tarkoituksena on asettaa pelimaailmaan peliobjekteja. Metodin toiminta perustuu annettuihin parametreihin spawnPosition ja blockID. SpawnPosition-parametri on Vector2-muodossa, joka tarkoittaa sitä että se säilöö sisällään kahta arvoa. Tässä tapauksessa nämä arvot viittaavat korkeuteen ja leveyteen mihin halutaan peliobjekti asettaa kun taas BlockID-parametri on numero joka viittaa mikä peliobjekti halutaan luoda listasta. Näillä tiedoilla metodi luo haluttuun positioon, halutun peliobjektin.

```
public void SpawnBlock(Vector2 spawnPosition, int blockID)
{
    GameObject block = (GameObject)Instantiate(
        blockPrefabs[blockID],
        spawnPosition,
        Quaternion.identity);

    block.transform.parent = this.transform;
}
```

Koodiesimerkki 10: SpawnBlock-metodi.

DestroyBlock-metodin (Koodiesimerkki 11) tarkoituksena on halutessa tuhota peliobjekti pelimaailmasta. Sen toimintaperiaate perustuu siihen, että sille annetaan parametrina peliobjekti joka halutaan tuhota. Mutta ennen tuhoamista metodi päivittää peliobjektia vastaavan kohdan kartasta ja koska toimintona on poistaa peliobjekti, päivitetään tämä vastaava kohta kartasta tyhjään, eli nollaan. Näin peli tietää että kyseinen kohta on nyt tyhjä. Tämä on pelin kannalta tärkeää kun siirrytään verkko-ominaisuuksiin, koska kartan täytyy pysyä ajan tasalla myös pelaajille jotka liittyvät peliin jälkeenpäin.

```
public void DestroyBlock(GameObject block)
{
    try
    {
        Vector2 destroyPosition = block.transform.position;
        int x = (int)(destroyPosition.x / Block.Width);
        int y = (int)(destroyPosition.y / Block.Height);

        map.Blocks[x, y] = 0;
        Destroy(block);
    }
    catch
    {
        Debug.Log("Can't destroy block!");
    }
}
```

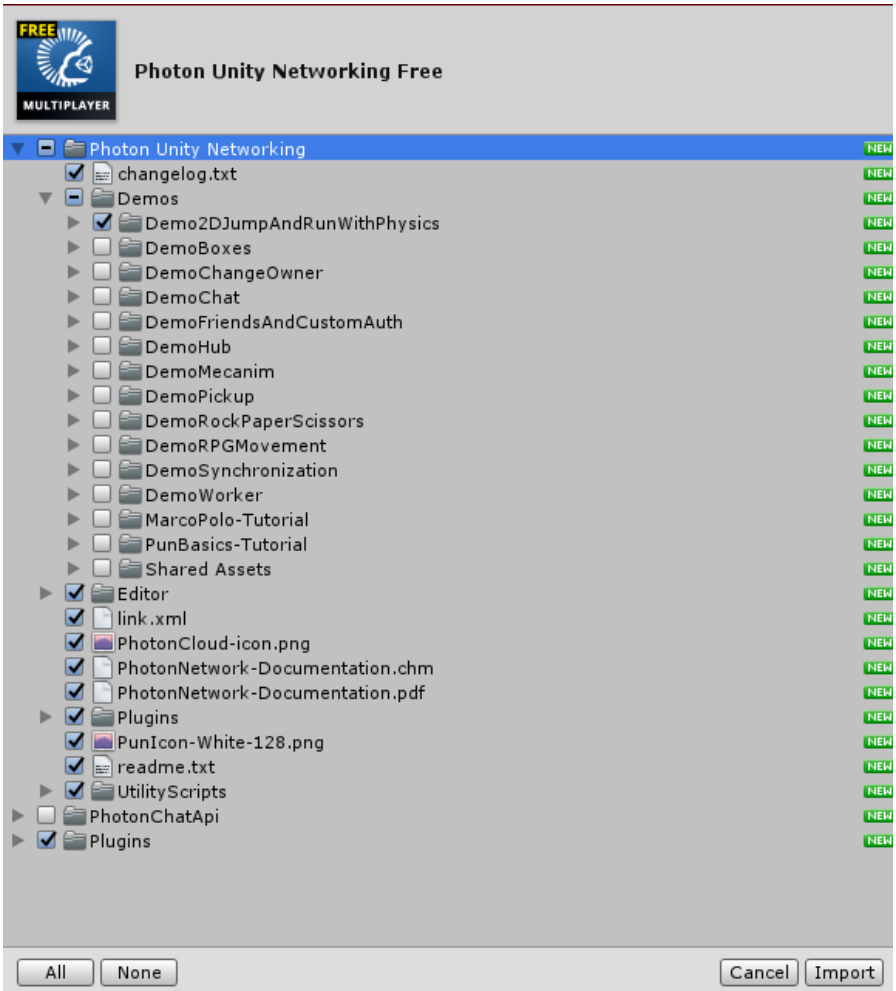
Koodiesimerkki 11: DestroyBlock-metodi.

6 VERKKOON SYNKRONOINTI

Verkkoon synkronoinnin olisi ollut parempi tehdä jo aikaisemman vaiheen yhteydessä, mutta päätin työssä kokeilla onnistuisiko verkkoon siirtyminen myös jälkeenpäin. Ajattelin myös, että työssä läpikäytävät asiat pysyisivät selvempinä minulle sekä lukijalle ja etteivät työn eri vaiheet tulisi liian sekavaksi ymmärtää. Valinnasta johtuen, jouduin tekemään muutoksia työssä aiemmin luotuihin peliobjekteihin sekä luokkiin.

6.1 PUN-työkalun liittäminen projektiin

Photon Unity Networkin (PUN) -työkalun liittäminen projektiin tapahtui lataamalla se ensin Unity Asset Storesta. Lataukseen pääsee käsiksi joko selaimella tai suoraan Unity-pelimoottorin sisällä hakemalla sitä nimellä Photon Unity Networkin Free. Latauksen jälkeen Unity kysyy haluaako käyttäjä rajata latauksen osia tarpeiden mukaan. Kuten kuvassa 10 näkyy, latasin vain ne osat joita tulen tarvitsemaan tässä opinnäytetyössä. Rajauksen jälkeen painetaan Import-painiketta jolloin Unity liittää paketin projektiin.



Kuva 11. Kuvakaappaus Asset-paketin liittämisestä.

Latauksen suoritettua, ilmestyy ruutuun PUN Setup -asennustyökalu, joka kysyy käyttäjältä sähköpostiosoitetta tai AppId:tä. Tämä tarkoitti sitä, että minun tarvitsi ensin rekisteröidä käyttäjätili työkalun verkkosivuilla, jonka jälkeen pääsin jatkamaan asennusta joko antamalla sähköpostiosoitteeni tai AppId:n jonka saa käyttöön rekisteröitymisen jälkeen työkalun verkkosivuilta. Tämän vaiheen jälkeen asennus oli valmis.

Asennuksen jälkeen sain käyttööni kaikki PUN-työkalun tarjoamat luokat, komponentit ja muut verkko-ominaisuudet, jotka mahdollistavat pelin päivityksen verkkoon sopivaksi.

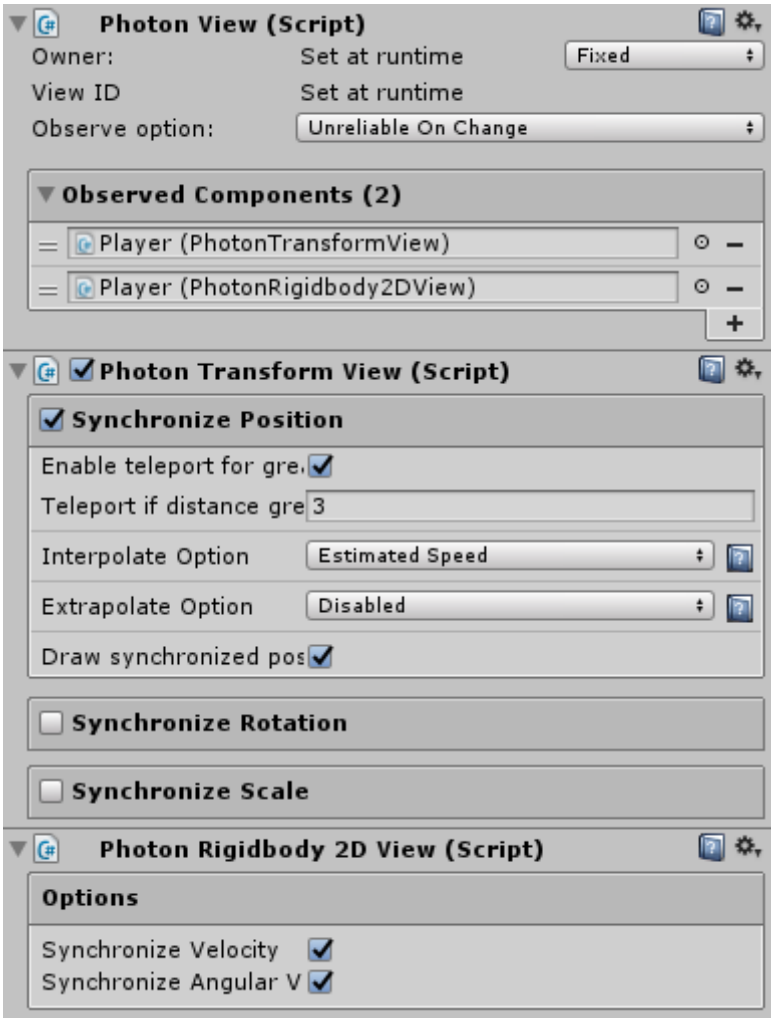
6.2 **Pelaaja peliobjekti**

Pelin verkko-ominaisuuksien yhtenä tavoitteena oli luoda peliin pelaajat, jotka pystyisivät halutessaan muokkaamaan ympärillä olevaa pelimaailmaa. Muokkaaminen tapahtuisi luomalla uusia peliobjekteja tai tuhoamaan jo olemassa olevia peliobjekteja pelin sisällä.

6.2.1 Komponentit ja skriptit

Päätin luoda pelaaja peliobjektin samalla tavalla jota käytin aikaisemmin luodessani peliobjekteja joista pelimaailma koostuu. Aluksi loin tyhjän peliobjektin nimellä Player ja lisäsin siihen Sprite Renderer -komponentin, antamaan sille ulkonäön, Box Collider 2D -komponentin, jotta se tunnistaisi törmäykset, sekä Rigidbody 2D -komponentin, joka antaisi sille painovoiman. Kyseiset komponentit ovat Unityn tarjoamia komponentteja, joita pystytään hyödyntämään myös verkossa.

Peruskomponenttien lisäksi tarvitsi peliobjektiin liittää PUN-työkalun tarjoamia komponentteja ja skriptejä. Ensimmäinen näistä oli Photon View -komponentti, joka seuraa ja päivittää muiden komponenttien ja skriptien toimintaa verkossa. Seuraavaksi lisäsin peliobjektiin PUN-työkalun mukana tulevan PhotonTransformView-skriptin, joka synkronoi automaattisesti peliobjektin sijainnin muille verkossa oleville. Skriptiin ei tarvinnut tehdä muutoksia vaan se oli valmis suoraan käytettäväksi. Lopuksi lisäsin peliobjektiin PhotonRigidbody2DView-skriptin seuraamaan peliobjektiin liitettyä RigidBody2D-komponenttia, synkronoiden automaattisesti painovoiman ja muut fysiikkamoottorin tiedot verkkoon. Jotta näiden skriptien viestintä onnistuisi verkossa, tarvitsi ne vielä lisätä kuvan 12 mukaisesti PhotonView-komponentin seurattavaksi.



Kuva 12. Kuvakaappaus pelaaja-peliobjektista.

6.2.2 Liikkuminen

PUN-komponenttien avulla pelaajan sijainti, fysiikka ja painovoima päivittyvät automaattisesti muille verkkohuoneessa oleville, joten pelaajan liikkumista varten ei tarvinnut tehdä muita verkkosynkronointeja.

Tein uuden skriptin nimellä JumpAndRunMovement ja liitin sen pelaaja-peliobjektiin (Koodiesimerkki 12). Skripti kuuntelee jatkuvasti käyttäjän antamia syötteitä ja niiden perusteella liikuttaa pelaajaa eri suuntiin. Rajoitin skriptin käytön ainoastaan paikalliselle käyttäjälle, koska muiden verkkohuoneessa olevien syötteet häiritsivät sen toimintaa. Muut käyttäjät saavat jo pelaajan tiedot PUN-komponenttien kautta, joten heidän ei tarvitse suorittaa skriptiä lainkaan.

```
public class JumpAndRunMovement : MonoBehaviour
{
    Rigidbody2D m_Body;
    public float Speed;
    public float JumpForce;

    void FixedUpdate()
    {
        if (GetComponent<PhotonView>().isMine == false)
        {
            return;
        }
        UpdateMovement();
    }

    void UpdateMovement()
    {
        if (m_Body == null)
        {
            m_Body = GetComponent<Rigidbody2D>();
        }
        Vector2 movementVelocity = m_Body.velocity;

        if (Input.GetButton("Jump"))
        {
            m_Body.AddForce(Vector2.up * JumpForce);
        }

        if ( Input.GetAxisRaw( "Horizontal" ) > 0.5f )
        {
            movementVelocity.x = Speed;

        }
        else if( Input.GetAxisRaw( "Horizontal" ) < -0.5f )
        {
            movementVelocity.x = -Speed;
        }
        else
        {
            movementVelocity.x = 0;
        }

        m_Body.velocity = movementVelocity;
    }
}
```

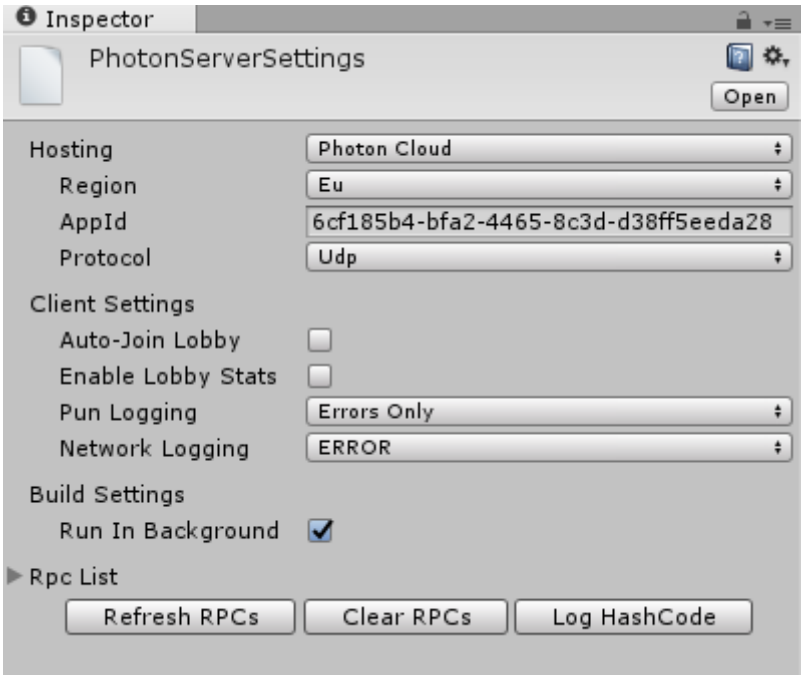
Koodiesimerkki 12. Pelaajan liikkuminen.

6.3 **Verkkohuoneet**

PUN-työkalun liittämisen ja pelaajan luomisen jälkeen tarvitsin vielä toiminnot PUN-verkon huoneiden tekemiseen ja niihin liittymiseen. Toiminnot pystytään helposti suorittamaan luomalla oma skripti käyttäen hyödyksi PUN-työkalun rajapintaa.

Käytin hyödyksi PUN-työkalun tarjoamaa ConnectAndJoinRandom-skriptiä. Skripti hakee automaattisesti asetukset yhteyden luomista varten PhotonServerSettings-tiedostosta (Kuva 13), joka on automaattisesti

luotu PUN-työkalun asennuksen yhteydessä. Asetuksista pystyy määrittämään verkon isännöitsijän, protokolan ja muita verkkoasetuksia.



Kuva 13. Kuvakaappaus PhotonServerSettings-tiedostosta.

Tässä työssä en suorittanut muutoksia tiedoston asetuksiin vaan jatkoin työtä luomalla peliin uuden tyhjän peliobjektin johon liitin ConnectAndJoinRandom-skriptin. Peliä ajettaessa skripti yhdistää pelin huoneeseen tai tekee uuden, jos sellaista ei löydy. Tämä ei kuitenkaan vielä tehnyt näkyvää muutosta peliin koska kaikki toiminnat tapahtuvat pelin taustalla.

Liitin samaan tyhjään peliobjektiin myös toisen skriptin nimellä OnJoinedInstantiate (Koodiesimerkki 12). Skripti sisältää OnJoinedRoom-metodin, jota PUN kutsuu pelaajan liittyessä huoneeseen. Metodi luo arvot Vector3-muuttujaan, siten että ne viittaavat koordinaattina pelimaailman keski-yläosaan. Koordinaatilla pelaaja peliobjekti luodaan verkkoon haluttuun positioon PhotonNetwork.Instantiate-metodilla. Metodin lopussa pelin kamera asetetaan luodun pelaajan ala-objektiksi, jolloin kamera seuraa pelaajaa sen liikkuessa.

```
public class OnJoinedInstantiate : MonoBehaviour
{
    public Vector3 spawnPos;
    public void OnJoinedRoom()
    {
        spawnPos = new Vector3(
            WorldGeneration.Instance.WorldWidth * Block.Width / 2,
            WorldGeneration.Instance.WorldHeight * Block.Height, 0);

        GameObject player = PhotonNetwork.Instantiate(
            "Player", spawnPos, Quaternion.identity, 0);

        Camera cam = Camera.main;
        cam.transform.SetParent(player.transform, false);

    }
}
```

Koodiesimerkki 13. OnJoinedInstantiate-skripti.

6.4 **Verkko-ominaisuuksien lisääminen luokkiin ja peliobjekteihin**

Tässä vaiheessa pelimaailma generoidaan lokaalisti ilman minkäänlaista synkronointia verkossa. Pelin käynnistyksen jälkeen, pelaajat liittyvät verkkohuoneeseen ja generoivat pelimaailman. Pelaajat ovat synkronoituna verkkoon aiempien PUN-komponenttien avulla ja heidän sijainti maailmassa päivittyy muille pelaajille mutta pelimaailmaa ei ole synkronoitu, jonka vuoksi pelaajat näkevät pelimaailman eri tavalla.

Muokkasin WorldGeneration-skriptiä lisäämällä siihen kaksi uutta metodia ja merkitsin ne PunRPC-merkinnöillä jolloin niitä voidaan kutsua verkossa PhotonView-komponentin kautta. Ensimmäiselle metodille annoin nimen RequestMap ja toiselle nimen SendMap.

RequestMap-metodia (Koodiesimerkki 13) kutsuttaessa se muuntaa pelimaailman kartan verkkoviestinnän kannalta pakolliseen byte-muotoon. Muuntamisen jälkeen metodi suorittaa RPC-toiminnon, kutsuen etäisesti SendMap-metodia, siten että, kaikki huoneessa olevat kutsuvat sitä, paitsi RPC-toiminnon suorittaja. RPC-toiminto lähettää byte-muotoon muunnetun kartan parametrina SendMap-metodille. SendMap-metodin (Koodiesimerkki 14) parametrina saatu kartta muunnetaan metodissa takaisin alkuperäiseen muotoon ja asetetaan uudeksi kartan arvoksi WorldGeneration-skriptiin. Näin huoneen isännöitsijä pystyy lähettämään tarpeen tullen kartan muille huoneissa oleville.

```
[PunRPC]
void RequestMap()
{
    var byteArray = Serialize(map.Blocks);
    Debug.Log(byteArray.Length);

    GetComponent<PhotonView>().RPC(
        "SendMap",
        PhotonTargets.Others,
        byteArray);
}
```

Koodiesimerkki 14. RequestMap-metodi.

```
[PunRPC]
void SendMap(byte[] newMap)
{
    var intArray = Deserialize<int[,]>(newMap);
    Debug.Log(intArray.GetLength(0) + " " + intArray.GetLength(1));
    map.Blocks = intArray;
    WorldDrawer.Instance.DrawWorld(map);
}
```

Koodiesimerkki 15. SendMap-metodi.

WorldGeneration-skriptissä generoitava kartta tuli luoda vain huoneen isännöitsijän kohdalla, joten muokkasin skriptin Awake-metodia poistamalla siitä GenerateMap- ja DrawWorld-kutsut. Siirsin kutsut uuteen metodiin OnJoinedRoom (Koodiesimerkki 15). Metodi suorittaa kutsut pelaajan liittyessä huoneeseen isännöitsijänä. Muussa tapauksessa huoneen isännöitsijälle lähetetään RPC-kutsu suorittamaan RequestMap-metodi. Käytännössä tämä tarkoittaa sitä, että isännöitsijä generoi maailman kun taas jälkeenpäin liittyvät pelaajat ohittavat generoimisen ja pyytävät valmiiksi generoitua karttaa isännöitsijältä. Huoneeseen liittyessä isännöitsijä kutsuu DrawWorld-metodia generoimisen jälkeen, ja muut pelaajat kartan pyytämisen jälkeen.

```
public static WorldGeneration Instance;
private WorldGeneration() { }

[SerializeField]
private int worldWidth;
[SerializeField]
private int worldHeight;

public string seed;
public bool useRandomSeed;

[Range(0, 100)]
public int randomFillPercent;

public Map map;

void Awake()
{
    Instance = this;
    map = new Map(WorldWidth, WorldHeight);
}
```

Koodiesimerkki 16. WorldGeneration-skriptin alkuosa.

6.5 Peliobjektien luominen ja tuhoaminen verkossa

Viimeisenä vaiheena pelissä oli pelimaailman muokkaaminen jälkeenpäin. Suoritin vaiheen lisäämällä pelaajille mahdollisuuden luoda ja tuhota peliobjekteja.

Lisäsin pelaaja-peliobjektiin uuden BlockHandler-skriptin (Koodiesimerkki 17). Skripti sisältää Update-metodin, jota pelimoottori kutsuu jatkuvasti ajon aikana. Pelaajan painaessa hiiren vasemmanpuoleista painiketta, tallentaa skripti painalluksen sijainnin ja lähettää sen DoPlaceBlock-metodille.

```
void Update()
{
    if (Input.GetKeyDown(KeyCode.Mouse1))
    {
        Vector2 cameraPos =
            Camera.main.ScreenToWorldPoint(Input.mousePosition);

        Vector2 roundedPos = new Vector2(
            Mathf.Round(cameraPos.x / Block.Width) * Block.Width,
            Mathf.Round(cameraPos.y / Block.Width) * Block.Width);

        GetComponent<PhotonView>().RPC(
            "DoPlaceBlock",
            PhotonTargets.All,
            (Vector2)roundedPos);
    }
    else if (Input.GetKeyDown(KeyCode.Mouse0))
    {
        Vector2 cameraPos =
            Camera.main.ScreenToWorldPoint(Input.mousePosition);

        Vector2 roundedPos = new Vector2(
            Mathf.Round(cameraPos.x / Block.Width) * Block.Width,
            Mathf.Round(cameraPos.y / Block.Width) * Block.Width);

        GetComponent<PhotonView>().RPC(
            "DoDestroyBlock",
            PhotonTargets.All,
            (Vector2)roundedPos);
    }
}
```

Koodiesimerkki 17. BlockHandler-luokan Update-metodi.

DoBlaceBlock-metodi (Koodiesimerkki 18) on merkattu RPC-metodiksi, jolloin sitä voidaan kutsua etäisesti. Metodia kutsutaan kaikkien verkkohuoneessa olevien kohdalla. Metodia kutsuttaessa se asettaa pelimaailmaan uuden peliobjektin parametrina saatuun sijaintiin. Jokainen pelaaja saa tiedon uudesta peliobjektista koska metodia kutsutaan jokaisen pelaajan kohdalla.

```
void DoPlaceBlock(Vector2 pos)
{
    WorldGeneration.Instance.map.Blocks[
        Mathf.RoundToInt(pos.x / Block.Width),
        Mathf.RoundToInt(pos.y / Block.Height)] = 1;

    GameObject block = (GameObject)Instantiate(
        Resources.Load("rockBlock"),
        pos,
        Quaternion.identity);
}
```

Koodiesimerkki 18. DoBlaceBlock-metodi.

Pelaajan painaessa hiiren oikeanpuoleista painiketta, skripti tallentaa painalluksen sijainnin ja lähettää sen DoDestroyBlock-metodille.

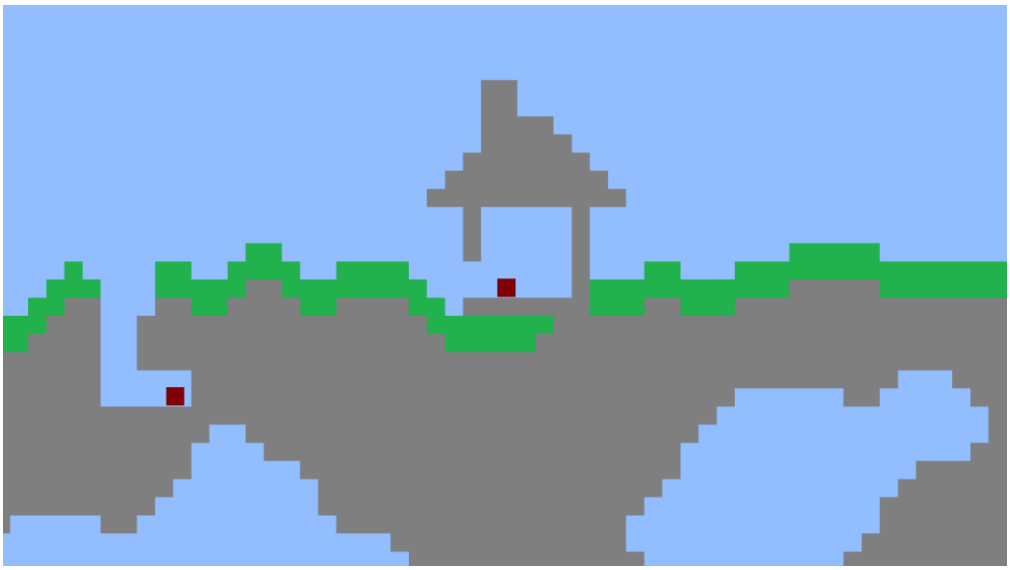
DoDestroyBlock-metodi (Koodiesimerkki 19) on merkitty RPC-metodiksi, jota kutsuttaessa peli tuhoaa peliobjektin, jos parametrina saadussa sijainnissa on peliobjekti.

```
[PunRPC]
void DoDestroyBlock(Vector2 pos)
{
    WorldGeneration.Instance.map.Blocks[
        Mathf.RoundToInt(pos.x / Block.Width),
        Mathf.RoundToInt(pos.y / Block.Height)] = 0;

    RaycastHit2D hit = Physics2D.Raycast(pos, Vector3.forward);
    if (hit.collider != null && hit.collider.tag != "Player")
    {
        Destroy(hit.transform.gameObject);
    }
}
```

Koodiesimerkki 19. DoDestroyBlock-metodi.

Metodien avulla pelaajat pystyvät halutessaan tuhoamaan ja luomaan peliobjekteja pelimaailmassa (Kuva 14).



Kuva 14. Kuvakaappaus generoidusta pelimaailmasta.

7 YHTEENVETO

Unity-pelimoottori on helppo ja kätevä vaihtoehto jopa aloittelevalla pelikehittäjälle. Moottori sopii sekä pienien että isojen pelien kehitykseen. Unityn omilta sivuilta löytyvä dokumentaatio ja manuaali olivat erittäin kattavia, ja suurin osa työn tarvittavasta tiedosta löytyi sen kautta, usein jopa kuvien ja videoiden kera.

Photon Unity Networking -työkalun integrointi Unity-pelimoottoriin onnistui helposti työkalun ohjeita seuraamalla. Työkalun ominaisuudet ovat hyvin läheisiä Unityn oman verkkorajapinnan kanssa, joten sen käyttö on yksinkertaista erityisesti vakiintuneille Unityn käyttäjille. Työkalusta löytyy myös todella kattava dokumentaatio ja sillä on aktiivinen ylläpito jatkuvasti vastaamassa kysymyksiin.

Pelimaailman generoimisen tavoitteena oli luoda ruudukkomainen, hyvin yksinkertainen pelimaailma, algoritmien mukaisesti. Generointi ja kaikki siihen liittyvät skriptit onnistuivat, ja kaikki ennalta määritetyt tavoitteet saavutettiin. Tavoitteisiin pääseminen oli mielestäni hyvän suunnitelman ja toteutustavan ansiota.

Pelin verkko-ominaisuuksien lisääminen olisi kannattanut tehdä jo pelimaailman generoimisen yhteydessä, koska jouduin palaamaan takaisin vanhoihin skripteihin ja muokkaamaan niitä Photon Unity Network -työkalun ominaisuuksilla. Aikaa olisi säästynyt, jos olisin lisännyt työkalun peliin jo aikaisemmassa vaiheessa ja rakentanut skriptit alusta asti verkko-ominaisuuksien ympärille.
Valmis tuotos saatiin aikaseksi ja ominaisuuksiltaan työ toimii pelipohjana, niin kuin tavoitteissa määriteltiin. Tarkoituksena on jatkaa työn kehitystä harrastemielessä. Kehitettävää on ainakin tekstuurien ja kokonaan puuttuvien äänien kohdalla. Huomasin myös jälkeenpäin, että työssä käytettävä tapa jakaa kaikki pelimaailman tiedot yhdellä RPC-metodilla, ei tule toimimaan, jos pelimaailmaa halutaan suurentaa. Pelimaailmaa suurennettaessa kasvaa sen rakentamiseen tarvittavat tiedot eksponentiaalisesti, jolloin riski ylikuormittumiseen kasvaa. Mahdollisena ratkaisuna ongelmaan on jakaa pelimaailma tiedot pienempiin osiin ja jakaa vain ne osat joita pelaajat tarvitsevat, kokonaisen pelimaailman sijaan.

LÄHTEET

Unity. 2017a. Unity3D. Haettu 17.1.2017 osoitteesta https://unity3d.com

Unity. 2016b. Public-relations. Haettu 17.1.2017 osoitteesta http://unity3d.com/public-relations

Unity Asset Store. 2017. Asset Store. Haettu 17.1.2017 osoitteesta https://www.assetstore.unity3d.com/en/

Unity Manual. 2016a. The Hierarchy Window. Haettu 17.1.2017 osoitteesta http://docs.unity3d.com/Manual/Hierarchy.html

Unity Manual. 2016b. The Project Window. Haettu 17.1.2017 osoitteesta http://docs.unity3d.com/Manual/ProjectView.html

Unity Manual. 2016c. The Inspector Window. Haettu 17.1.2017 osoitteesta http://docs.unity3d.com/Manual/UsingTheInspector.html

Unity Manual. 2016d. Using Components. Haettu 17.1.2017 osoitteesta http://docs.unity3d.com/Manual/UsingComponents.html

Unity Manual. 2016e. Console Window. Haettu 17.1.2017 osoitteesta http://docs.unity3d.com/Manual/Console.html

Rigidbody. 2016. Haettu 17.1.2017 osoitteesta http://docs.unity3d.com/Manual/class-Rigidbody2D.html

C# vs Unityscript – Which is faster? Haettu 17.1.2017 osoitteesta http://dentedpixel.com/developer-diary/c-vs-unityscript-which-is-faster/

Photon Unity Networking (2017a). Unity Networking. Code Examples. Haettu 17.1.2017 osoitteesta https://www.photonengine.com/en/PUN

Photon Unity Networking (2017b). Unity Networking. Photon Cloud - The Power Behind. Haettu 17.1.2017 osoitteesta https://www.photonengine.com/en/PUN

Photon Unity Networking (2017c). Unity Networking. Scalable, Reliable & Always Connects. Haettu 17.1.2017 osoitteesta https://www.photonengine.com/en/PUN

Photon Unity Networking (2017d). Feature Overview. Haettu 24.1.2017 osoitteesta http://doc.photonengine.com/zh-cn/pun/current/getting-started/feature-overview

Cellular Automata. 2016. Haettu 17.1.2017 osoitteesta https://unity3d.com/learn/tutorials/projects/procedural-cave-generation-tutorial/cellular-automata?playlist=17153