Unity ja UNET -moninpeliprojekti

Joni Lähdesluoma

Opinnäytetyö
Elokuu 2015
Tietotekniikka
Ohjelmistotekniikka

TIIVISTELMÄ

Tampereen ammattikorkeakoulu
Tietotekniikka
Ohjelmistotekniikka

Lähdesluoma Joni
Unity ja UNET -moninpeliprojekti

Opinnäytetyö 39 sivua, joista liitteitä 9 sivua
Marraskuu 2015

Työn tarkoituksena oli tutustua Unityn uusiin moninpelityökaluihin ja luoda toimiva prototyyppi neljän hengen verkkopelistä näitä työkaluja hyödyntäen.

Ensin tarkastellaan Unityn tarjoamia palveluita liittyen moninpelien tekoon ja puhutaan hieman Asset Storesta.

Tämän jälkeen käydään läpi Unityn uudet moninpeli työkalut ja API nimeltä UNET. Tämä työ käy läpi UNETin tuomat uudet Unity-komponentit ja APIn luokat ja kuinka niitä käytetään moninpelin teossa.

Lopuksi tutustutaan prototyypin tärkeimpiin kohtiin ja kuinka ne toimivat.

Asiasanat: Unity, pelinkehitys, moninpeli, verkkopeli

ABSTRACT

Tampereen ammattikorkeakoulu
Tampere University of Applied Sciences
Degree programme in ICT Engineering
Software Engineering

Lähdesluoma Joni
Unity and UNET in multiplayer game project

Bachelor's thesis 39 pages, appendices 9 pages
November 2015

The purpose of this thesis was to learn to use Unity's new networking tools and services by creating a simple multiplayer game prototype.

We will first take a look at some of Unity's services related to making multiplayer games as well as talk a little about Asset Store.

Then we will talk about Unity's new networking tools and API called Unity Networking or UNET for short. This thesis goes through the most important new components and classes required for simple multiplayer games as well as talks about their use.

In the end we will go through the most important parts of the game prototype and take a look at how they work.

Key words: Unity, development, multiplayer, games

SISÄLLYS

1 JOHDANTO 5
2 MONINPELI 6
3 UNITY-PELIMOOTTORI 8
3.1 Asset Store -kauppapaikka 8
3.2 Internetpalvelut 9
4 UNET-työkalut 10
4.1 Uudet komponentit ja luokat 10
4.1.1 NetworkIdentity-komponentti 10
4.1.2 NetworkBehaviour-komponentti 11
4.1.3 NetworkManager-komponentti 11
4.1.4 NetworkTransform-komponentti 12
4.1.5 NetworkServer-komponentti 12
4.1.6 NetworkClient-komponentti 13
4.1.7 NetworkAnimator-komponentti 13
4.2 Matalan tason LLAPI-rajapinta 13
4.3 Korkean tason HLAPI-rajapinta 14
4.3.1 Isäntä ja asiakas 14
4.3.2 Pelaajat ja auktoriteetti 15
4.3.3 Etäproseduurikutsut (RPC) 16
4.3.4 Pelitilanteen synkronointi 18
4.4 Multiplayer Lobby -järjestelmä 20
4.4.1 Multiplayer Lobby Manager -komponentti 21
4.4.2 Multiplayer Lobby Player ja Game Player 21
5 PELIPROJEKTI 22
5.1 Aula 23
5.2 Peli 26
5.2.1 Pelaajan liikkuminen 26
5.2.2 Ampuminen 28
5.2.3 Törmäyksen tunnistus 29
6 POHDINTA 30
LÄHTEET 31
LIITTEET 32
Liite 1. Pelaajan liikkuminen 32
Liite 2. Ampuminen 35
Liite 3. Ammuksen törmäyksen tunnistus 37
Liite 4. Pelaajan elämän hallinta 39

1 JOHDANTO

Tässä työssä käydään läpi Unityn UNET-tekniikan toimintaa ja sen tuomia uusia komponentteja. Luvussa 2 tutustutaan lyhyesti Unityn viimeaikaisiin uudistuksiin sekä sen erilaisiin palveluihin. Luvussa 3 käydään läpi UNET, sen toiminta ja sen tuomat uudistukset sekä puhutaan lyhyesti Unityn Multiplayer palvelusta. Tämän lisäksi tarkastellaan paremmin UNETin korkean tason APIa ja sen käyttöä moninpeleissä. Luvussa 4 käydään läpi lyhyesti moninpeliprototyypin tärkeimmät osat ja luodaan katsaus sen koodiin ja komponentteihin.

Yksi vaikeimmista asioista toteuttaa Unitylla on ollut moninpeli. Vanha API vaati paljon koodia ja suuren määrän tietoa verkkopelien toteutuksesta. UNET tuo mukanaan täysin uuden APIn ja uusia komponentteja joilla moninpelin luominen on huomattavasti helpompaa eikä vaadi aiempaa kokemusta verkkopelien teosta.

2 MONINPELI

Moninpeli on pelityyppi, jossa kaksi tai useampi pelaaja pelaavat samanaikaisesti samassa pelissä. Tämä voi tapahtua verkon yli tai samalla laitteella. Pelistä riippuen pelaajat voivat pelata toisiaan vastaan tai yrittää saavuttaa yhteisen tavoitteen.

Ensimmäiset moninpelit toimivat vertaisverkko-mallin mukaisesti. Jokainen pelaaja toimi sekä palvelimena ja asiakkaana ja jakoi tietoa kaikille muille pelaajille. Tässä mallissa on kuitenkin lukuisia ongelmia. Pelitilanteen pitäminen identtisenä kaikkien pelaajien kesken on erittäin hankalaa ja pienikin muutos saa aikaan sen, että pelitilanne ei ole enää identtinen. Koska kaikki pelaajat ovat samanarvoisia vertaisverkossa, oikean pelitilanteen löytäminen on mahdotonta ja pelin pelaamisesta tulee myös mahdotonta.

Näiden ongelmien korjaamiseksi siirryttiin käyttämään palvelin/asiakas –yhteyttä, jossa kaikki pelaajat ovat asiakkaita ja keskustelevat vain palvelimen kanssa. Kaikki peliin liittyvät toimet tehtiin palvelimella ja asiakkaat lähettivät tälle vain näppäinten ja hiiren painallukset. Palvelin laski näiden pohjalta uuden pelitilanteen ja lähetti sen tiedot kaikille asiakkaille. Tämä järjestelmä toimi hyvin, kunhan latenssi asiakkaan ja palvelimen välillä pysyi pienenä. Tämä korjaamiseksi alettiin asiakkaat eivät voineet enää vain lähettää näppäinten painalluksia, vaan niiden täytyi suorittaa osaa pelin koodista paikallisesti. Näin asiakkaat voitiin ohjelmoida ennustamaan pelaajan liikkeitä näppäinten painallusten ja aikaisempien liikkeiden mukaan. Tämä pienensi latenssin vaikutusta, koska asiakkaan ei tarvinnut enää odottaa näppäimen painalluksen lähettämistä palvelimelle ja palvelimen vastausta.

Asiakkaan ennustamisen tueksi tarvittiin vielä tapa korjata tilanteet joissa asiakas on väärässä. Asiakas kirjoittaa käyttäjän näppäinten painallukset ja niiden tapahtuma-ajan puskuriin ja käyttää näitä komentoja ennustaessaan tulevan pelitilanteen. Alkutilanteena ennustukselle asiakas käyttää viimeisintä palvelimelta tullutta tietoa pelitilanteesta. Ristiriitatilanteessa asiakas ottaa palvelimelta tulleen uuden tiedon ja poistaa puskurissa olevat tiedot, jotka ovat vanhempia kuin palvelimelta tullut tieto. Tämän jälkeen asiakas laskee pelitilanteet palvelimen pelitilanteesta taaksepäin viimeisimpään suoritettuun pelitilanteeseen. Näin vältytään tilanteelta, jossa pelaaja tuntuu hypähtävän, kun pelitilanne muuttuu äkisti väärästä ennustuksesta palvelimen oikeaan pelitilanteeseen. (Gaffer on Games)

3 UNITY-PELIMOOTTORI

Unity on Unity Technologiesin kehittämä pelimoottori, jonka ensimmäinen versio julkaistiin vuonna 2005. Viimeisin versio kirjoitushetkellä on 5.2.3. Työssä käytettiin versiota 5.2.2f1.

Unity julkaisi 2D-pelien tekoon tarkoitetut työkalut 4.3 päivityksessä, mikä mahdollisti 2D-pelien tekemisen natiiveilla työkaluilla. Tätä ennen 2D-pelien tekeminen oli hankalaa ilman Asset Storesta ostettuja työkaluja. (Unity uutissivut)

Unity 5 julkaisun yhteydessä Unity poisti rajoitukset ilmaisen version käyttäjiltä, mikä antoi ennen maksullisen lisenssin takana olleet työkalut kaikkien käyttöön. Unity 5 tarjosi myös uusia työkaluja ja 64-bittisen version ohjelmasta. (Unityn blogi)

5.1 versio toi mukanaan uuden verkkopelityökalut nimeltään UNET. (Unityn blogi)

3.1 Asset Store -kauppapaikka

Unity Asset Store on kauppapaikka, josta on mahdollista hankkia ilmaista tai maksullista sisältöä omiin projekteihin. Kaupasta löytyy valmiita 3D-malleja, animaatioita, tekstuureja, skriptejä, ääniä ja kokonaisia lisäosia Unity Editoriin.

Tässä projektissa on käytetty pohjana Unity Technologies julkaisemaa Network Game Lobby (beta) -pakettia, joka tarjoaa yksinkertaisen lobby-järjestelmäesimerkin. Tämä paketti käyttää hyväkseen uusia UNET-työkaluja. (Unity Asset Store)

Lobby eli aula on useissa verkkopeleissä esiintyvä tila, johon pelaajat liittyvät ennen kuin varsinainen peli alkaa. Usein tämä on vain eräänlainen valikko, jossa pelaaja voi tarkastella pelaajakohtaisia asetuksia ja nähdä tulevan pelin asetukset.

3.2 Internetpalvelut

Pelien kannalta ehkä tärkein Unityn tarjoama palvelu on matchmaking-palvelu, joka mahdollistaa pelaajien ja pelien löytämisen ilman IP-osoitteen syöttämistä. Verkkoliikenne ohjataan välityspalvelimen kautta, mikä välttää ongelmia palomuurien ja osoitteenmuutosten eli NAT-tekniikan kanssa. Välityspalvelin on Unityn tarjoama pilvipalvelu, joka toimii läheisesti matchmaking-palvelun kanssa. (Unity Manual)

Nämä palvelut erottavat Unityn sen tärkeimmästä kilpailijasta, Unreal Engine 4 -pelimoottorista. UE4 ei tarjoa valmista matchmaking-ratkaisua, osoitteenmuuntajan läpäisyä tai lobby-järjestelmää vaan nämä on käyttäjän tehtävä itse.

4 UNET-työkalut

UNET eli Unity Networking on Unity:n tarjoama kokoelma työkaluja ja palveluita verkkopelien tekemiseen. Se julkaistiin Unity 5.1 version yhteydessä. UNET on täysin uusi ratkaisu, joka korvaa vanhan verkkoratkaisun.

UNET voidaan jakaa karkeasti korkeaan ja matalaan tasoon, sen mukaan kuinka lähellä laitteistoa työskennellään. (Unity Manual)

4.1 Uudet komponentit ja luokat

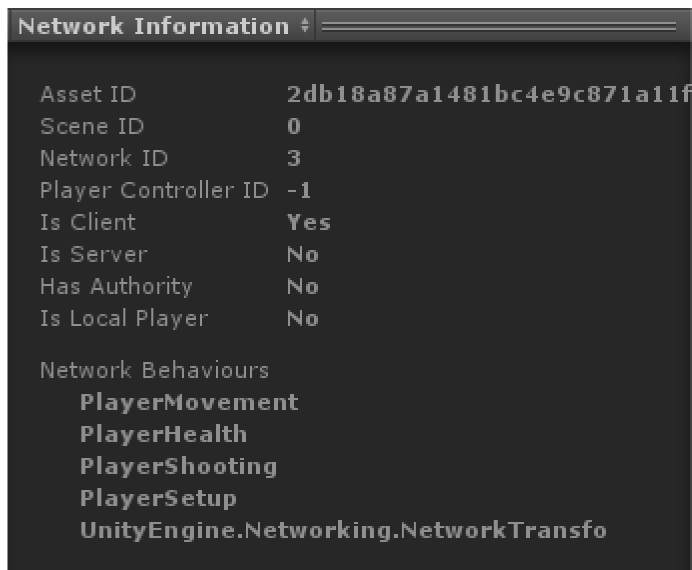
UNET toi mukanaan suuren määrän uusia komponentteja ja luokkia. Tässä työssä keskityttiin vain korkean tason APIn luokkiin ja komponentteihin.

4.1.1 NetworkIdentity-komponentti

NetworkIdentity on komponentti, joka on UNETin selkäranka. Se hallitsee objektin verkkoidentiteettiä ja tiedottaa itsestään verkkojärjestelmälle. Nämä objektit täytyy luoda palvelimelle käyttäen NetworkServer.Spawn()-funktiota. Tämän jälkeen ne saavat NetworkInstanceId-tunnisteen ja ne voidaan luoda asiakkaissa, jotka ovat yhteydessä palvelimeen.

Näyttämöllä eli pelialueella valmiina olevat objektit ovat valmiiksi sekä palvelimella että asiakkailla. Tämän vuoksi kaikki objektit joilla on NetworkIdentity-komponentti otetaan pois käytöstä. Kun asiakas ottaa yhteyttä palvelimeen, palvelin lähettää tiedot objekteista, jotka pitää ottaa käyttöön ja mitkä niiden uusimmat tiedot ovat. Tämä varmistaa että asiakkaat eivät aseta objekteja vääriin paikkoihin tai että ne eivät luo objekteja jotka on jo tuhottu.

Jokainen NetworkIdentity-komponentti sisältää seuranta-tietoa, kuten SceneID, NetworkID ja AssetID, joiden avulla samaa elementtiä käyttävät objektit tunnistetaan toisistaan. Alla oleva kuva näyttää mitä tietoja objektista saadaan ajon aikana. (Unity Manual)



KUVA 1. NetworkIdentity-komponentin tiedot ajon aikana.

4.1.2 NetworkBehaviour-komponentti

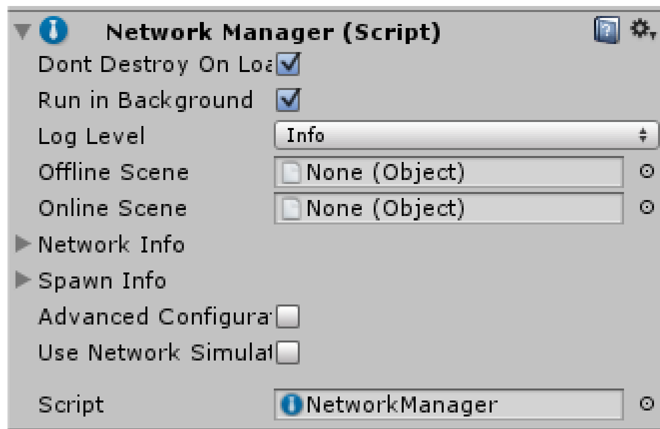
NetworkBehaviour on termi skripteille, jotka perivät NetworkBehaviour-luokan ja toimivat NetworkIdentity-komponentin kanssa. Nämä skriptit voivat käyttää UNETin korkean tason APIn funktioita.

NetworkBehaviour-skriptit mahdollistavat muuttujien synkroinoinnin palvelimelta asiakkaille, erilliset palvelin- ja asiakasfunktiot sekä kutsujen lähettämisen palvelimelta asiakkaille ja toisin päin. (Unity Manual)

4.1.3 NetworkManager-komponentti

NetworkManager-komponenttia käytetään pelitilanteen hallitsemiseen verkkopelissä. Tämä komponentti tarjoaa oman käyttöliittymän editorissa, joten sitä voi periaatteessa käyttää täysin ilman koodausta. Se on rakennettu käyttäen UNETin korkea tason APIa ja se voidaan korvata omalla skriptillä. Sen tarkoitus on tarjota helppokäyttöinen ratkaisu moninpelin hallintaan.

NetworkManager on suunniteltu niin, että sen toimintaa voi muuttaa käyttäen luokan virtuaalifunktioita. Näin on myös tässä työssä käytetyssä Network Lobby Manager -komponentissa. (Unity Manual)

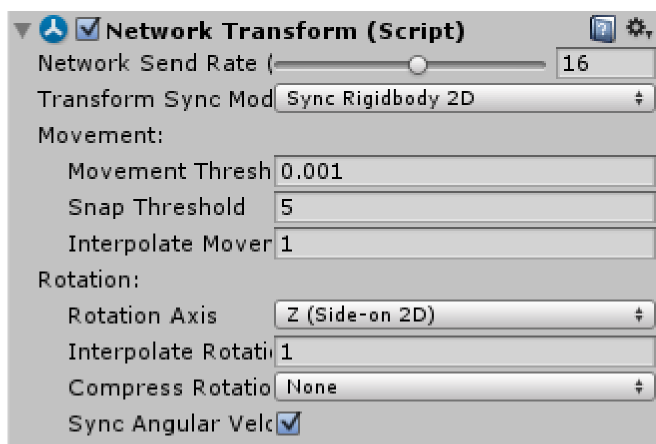


KUVA 2. NetworkManager-komponentti

4.1.4 NetworkTransform-komponentti

NetworkTransform-komponenttia käytetään liikkeen synkronoimisessa muille pelaajille. Tämä komponentti ottaa huomioon objektin auktoriteetin. Jos objektilla on paikallinen auktoritetti, engl. local authority, sen sijainti synkronoidaan palvelimelle ja sen kautta muille pelaajille. Muussa tapauksessa objektin sijainti synkronoidaan palvelimelta asiakkaille.

Tämä komponentti vaatii että objektilla on NetworkIdentity-komponentti. (Unity Manual)



KUVA 3. NetworkTransform komponentti.

4.1.5 NetworkServer-komponentti

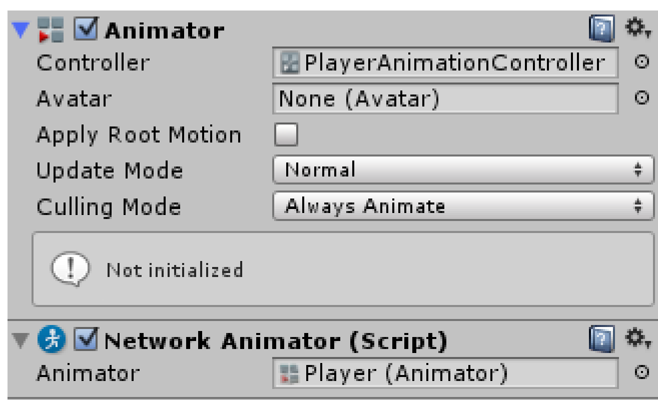
NetworkServer on UNETin korkean tason APIn luokka, jonka tehtävänä on hallita yhteyksiä usealta asiakkaalta samaan aikaan. (Unity Manual)

4.1.6 NetworkClient-komponentti

NetworkClient on korkean tason APIn luokka, joka hallitsee yhteyttä palvelimeen. Sitä voidaan käyttää lähettämään ja vastaanottamaan viestejä palvelimelta. Se auttaa myös hallitsemaan spawnattuja objekteja sekä reitittämään etäproseduurikutsuja. (Unity Manual)

4.1.7 NetworkAnimator-komponentti

NetworkAnimator komponentti synkronoi animaatiot verkon yli. Se ottaa vastaan Animator-komponentin ja pitää tämän ajamat animaatiot synkronoituna verkon yli. (Unity Manual)



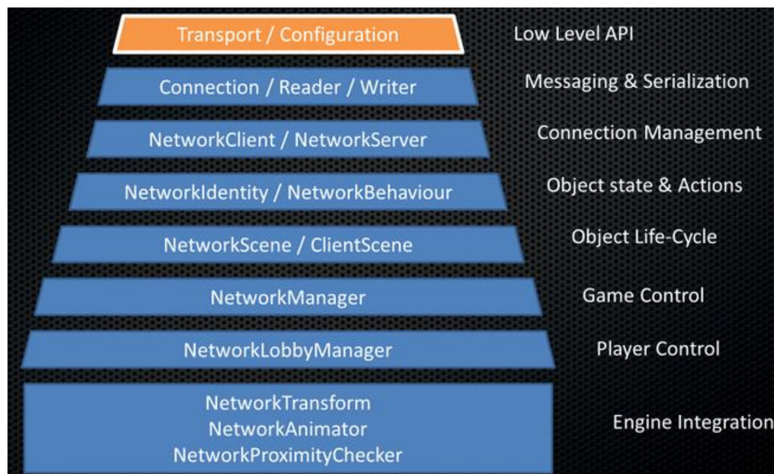
KUVA 3. Animator- ja Network Animator -komponentit.

4.2 Matalan tason LLAPI-rajapinta

Matala taso käyttää Transport Layer APIa, joka toimii käyttöjärjestelmän soketti-pohjaisen verkon päällä. Se on tarkoitettu edistyneempiin verkkopeleihin, jotka tarvitsevat yksilöllisempiä ratkaisuja. Tässä työssä ei käytetty suoraan Transport Layer APIa, vaan korkea taso valittiin sen helppokäyttöisyyden ja yksinkertaisuuden vuoksi. (Unity Manual)

4.3 Korkean tason HLAPI-rajapinta

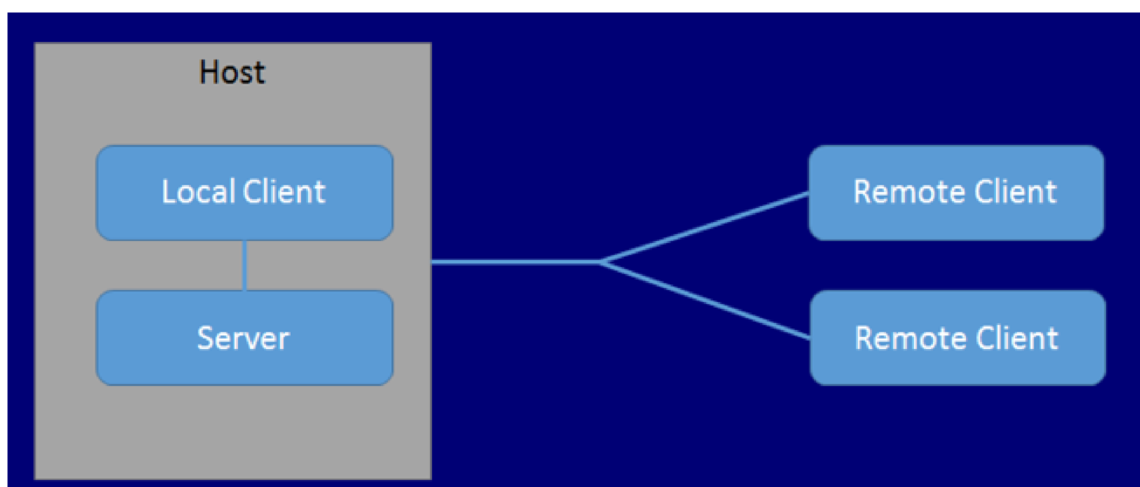
Korkea taso eli High Level API on rakennettu matalan tason päälle ja se automatisoi monia sen tehtäviä. HLPAI on auktoritatiivinen palvelinjärjestelmä. Palvelimella on täysi määräysvalta ja asiakkaat ovat niin sanottuja tyhmiä asiakkaita (dumb client). Ne lähettävät tietoa palvelimelle, joka päättää mitä tiedon pohjalta tapahtuu ja lähettää ratkaisunsa takaisin asiakkaille. (Unity Manual)



KUVA 4. HLAPI muodostuu useasta eri tasosta (Unity Manual)

4.3.1 Isäntä ja asiakas

UNET järjestelmässä peleillä on yksi palvelin ja monta asiakasta. Yksi asiakkaista toimii myös palvelimena, joten erillistä palvelinsovellusta ei tarvita.

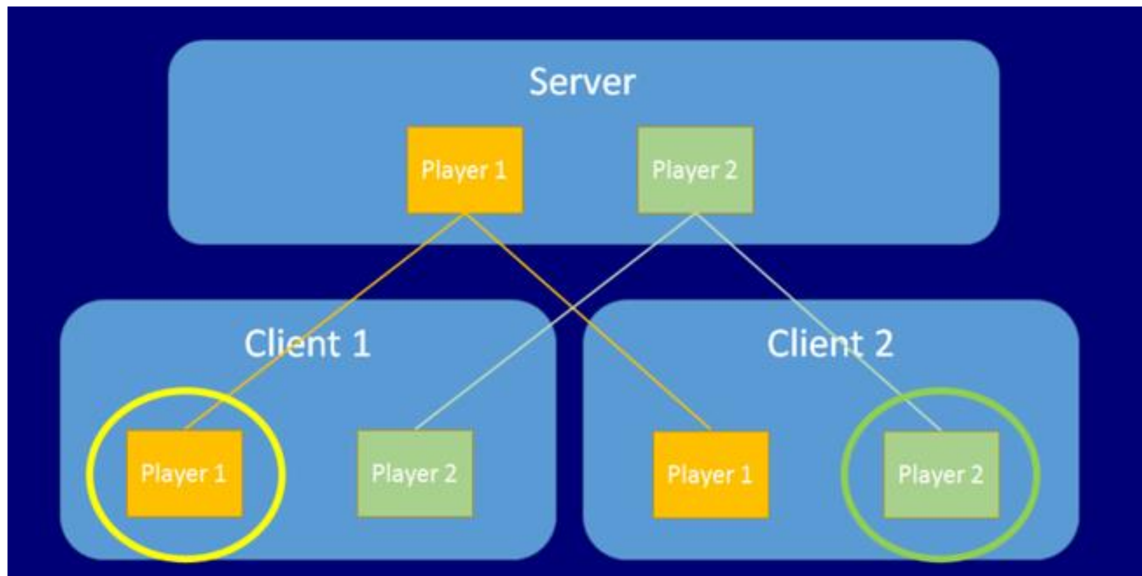


KUVA 5. UNET-verkon toiminta (Unity Manual)

Palvelimena toimivaa asiakasta kutsutaan isännäksi. Isäntä eroaa etäasiakkaista siten, että se keskustelee palvelimen kanssa suoraan funktiokutsuilla, kun taas etäasiakkaat viestittävät normaalisti verkon yli. Tämä järjestelmä on suunniteltu siten, että isäntä ja etäasiakas jakavat saman koodin, jotta pelinkehittäjän ei tarvitse ajatella kuin yhtä asiakastyyppiä. (Unity Manual)

4.3.2 Pelaajat ja auktoriteetti

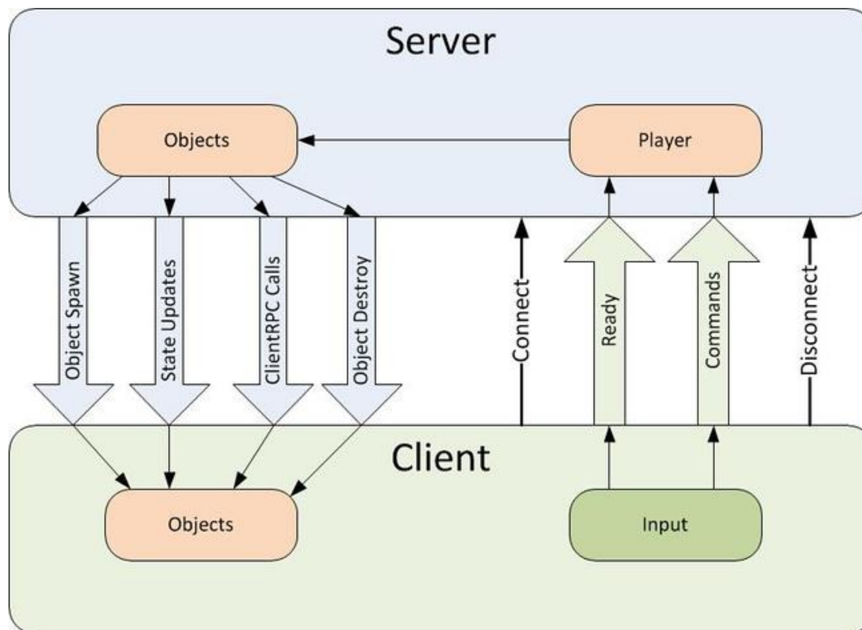
Pelaajan ja pelattavan olion (esim. pelihahmo) välillä on liitos, joka merkitsee olion pelaajan omaksi. Muut pelaajat eivät voi suorittaa komentoja tälle oliolle. Kaikki keskustelu tapahtuu olion ja palvelimen välillä. Tästä oliosta käytetään termiä local player ja sillä on ominaisuus isLocalPlayer, jolle asetetaan arvo true. (Unity Manual)



KUVA 6. Kaksi asiakasta ja niiden local player –oliot (Unity Manual)

4.3.3 Etäproseduurikutsut (RPC)

UNET käyttää kahdentyyppisiä etäproseduurikutsuja: Command-kutsua, jossa asiakas kutsuu ja palvelin suorittaa palvelun ja ClientRpc-kutsua, jossa palvelin kutsuu ja asiakas suorittaa palvelun. Alla oleva kuva ilmentää mihin suuntaan etäkutsut kulkevat.



KUVA 7. Etäkutsujen kulkusuunta. (Unity Manual)

Command-kutsut lähetetään asiakkaan omistamalta pelaaja-objektilta palvelimella olevalle asiakkaan pelaaja-objektille. Turvallisuussyistä asiakas voi lähettää kutsun vain omistamaansa objektia vastaavalle objektille. Tämä estää muiden omistamien objektien hallitsemisen.

Funktion muuttamiseksi Command-kutsuksi sitä ennen lisätään [Command]-atribuutti ja siihen lisätään Cmd-etuliite. Huomaa, että tämä koodi suoritetaan palvelimella. Alla on selvyydeksi esimerkkikoodi. (Unity Manual)

```
class Player : NetworkBehaviour
{

    public GameObject bulletPrefab;

    [Command]
    void CmdDoFire(float lifeTime)
    {
        GameObject bullet = (GameObject)Instantiate(
            bulletPrefab,
            transform.position + transform.right,
            Quaternion.identity);

        var bullet2D = bullet.GetComponent<Rigidbody2D>();
        bullet2D.velocity = transform.right * bulletSpeed;
        Destroy(bullet, lifeTime);

        NetworkServer.Spawn(bullet);
    }

    void Update()
    {
        if (!isLocalPlayer)
            return;

        if (Input.GetKeyDown(KeyCode.Space))
        {
            CmdDoFire(3.0f);
        }

    }
}
```

KUVA 8. Koodiesimerkki Command-kutsusta (Unity Manual)

Kuvan 8 esimerkissä käytetään Command-kutsua luomaan ammus, jos pelaaja painaa välilyötiä. Tämä Command-funktio ottaa vastaan panoksen eliniän, jonka jälkeen panos luodaan valmiin elementin mukaan. Panos asetetaan tuhoutumaan eliniän kuluttua. Lopuksi panos luodaan käyttäen Network.Spawn()-funktiota. Tämä lähettää käskyn asiakkaille luoda sama panos samaan paikkaan. Näin asiakas näyttää saman tilanteen kuin palvelinkin.

ClientRpc-kutsut lähetetään palvelimella olevalta objektilta. Niitä pystyy lähettämään mikä tahansa palvelimella oleva objekti, jolla on NetworkIdentity-komponentti ja joka on luotu käyttäen Network.Spawn()-funktiota. Koska palvelimella on täysi määräysvalta (auktoritatiivinen järjestelmä) ClientRpc-kutsuihin ei sisälly turvallisuusriskiä.

Funktio muutetaan ClientRpc-kutsuksi lisäämällä sitä ennen [ClientRpc]-atribuutti ja lisäämällä Rpc-etuliite. Alla oleva esimerkkikoodi näyttää kuinka tämä tapahtuu käytännössä. (Unity Manual)

```
class Player : NetworkBehaviour
{

    [SyncVar]
    int health;

    [ClientRpc]
    void RpcDamage(int amount)
    {
        Debug.Log("Took damage:" + amount);
    }

    public void TakeDamage(int amount)
    {
        if (!isServer)
            return;

        health -= amount;
        RpcDamage(amount);
    }
}
```

KUVA 9. ClientRpc esimerkkikoodi (Unity Manual)

Kuvassa 9 oleva TakeDamage()-funktio tarkistaa aluksi onko skriptin suorittaja palvelin. Vain palvelin voi suorittaa ClientRpc-kutsuja. Jos skriptin suorittaja ei ole palvelin, funktion suoritus lopetetaan. Tarkistuksen jälkeen vähennetään health-muuttujan arvosta vahingon määrä ja RpcDamage()-funktio saa tämän vahinkoarvon argumenttina. RpcDamage()-funktio tulostaa vahinkoarvon konsoliin.

4.3.4 Pelitilanteen synkronointi

Synkronointi tapahtuu palvelimelta etäasiakkaille. Isäntä ei tarvitse sitä, koska se toimii paikallisena asiakkaana ja palvelimena ja jakaa datan tämän kanssa. Ainoa poikkeus on SyncVar-koukut, jotka suoritetaan myös paikallisen asiakkaan puolella.

SyncVar:it ovat NetworkBehaviour-skriptin jäsenmuuttujia, jotka synkronoidaan palvelimelta asiakkaalle. Kun objekti luodaan Network.Spawn()-funktiolla tai pelaaja liittyy jo käynnissä olevaan peliin, tälle lähetetään uusin tieto SyncVar muuttujista, jotka kyseinen objekti voi nähdä. Jäsenmuuttuja muutetaan SyncVar-muuttujaksi lisäämällä sitä ennen [SyncVar]-atribuutti. Alla tästä esimerkki. (Unity Manual)

```
class Player : NetworkBehaviour
{

    [SyncVar]
    int health;

    public void TakeDamage(int amount)
    {
        if (!isServer)
            return;

        health -= amount;
    }
}
```

KUVA 10. SyncVar-muuttujan esimerkkikoodi. (Unity Manual)

SyncVar-muuttujat voivat olla tyypiltään perustyyppejä sekä Unityn omia tietotyyppejä ja käyttäjän luomia struct-tietotyyppejä. Yhdessä skriptissä voi olla 32 SyncVar-muuttujaa mukaan lukien SyncList listat. (Unity Manual)

SyncList toimii kuten SyncVar-muuttuja, mutta on nimensä mukaisesti lista. Ne eivät tarvitse erillistä [SyncVar]-atribuuttia, vaan ovat omia luokkiaan. Jokaisella perustyypillä on oma luokkansa.

- SyncListString on lista String-tyyppisiä arvoja
- SyncListFloat on lista liukulukuja
- SyncListInt on lista kokonaislukuja
- SyncListUInt on lista positiivisia kokonaislukuja
- SyncListBool on lista bool-arvoja

Näiden lisäksi on olemassa SyncListStruct, joka antaa käyttäjälle mahdollisuuden luoda omia tietuita. Tämä tietue voi sisältää vain perustyyppejä sekä Unityn omia tietotyyppejä. (Unity Manual)

SyncList listat sisältävät SyncListChanged-delegaatin Callback, joka tiedottaa asiakkaita, kun listan sisältöä muutetaan. Alla oleva esimerkki näyttää miten sitä käytetään. (Unity Manual)

```
public class MyScript : NetworkBehaviour
{
    public struct Buf
    {
        public int id;
        public string name;
        public float timer;
    };

    public class TestBufs : SyncListStruct<Buf> {}
    TestBufs m_bufs = new TestBufs();

    void BufChanged(Operation op, int itemIndex)
    {
        Debug.Log("buf changed:" + op);
    }

    void Start()
    {
        m_bufs.Callback = BufChanged;
    }
}
```

KUVA 11. Callback-delegaatin käyttö. (Unity Manual)

4.4 Multiplayer Lobby -järjestelmä

Useassa moninpelissä on eräänlainen aula, johon pelaajat liittyvät ennen varsinaisen pelin pelaamista. Tässä aulassa pelaajat voivat mahdollisesti muokata asetuksia ja lopulta asettaa itsensä valmiiksi. Peli alkaa kun kaikki pelaajat ovat valmiina. (Unity Manual)

4.4.1 Multiplayer Lobby Manager -komponentti

MultiplayerLobbyManager on muokattu versio NetworkManager komponentista, joka luo aulan johon pelaajat liittyvät ennen pelin alkamista. Se antaa kehittäjän valita seuraavat asetukset aulalle:

- Pelaajien maksimimäärän
- Alkaako peli kun kaikki ovat valmiina
- Pystyykö peliin liittymään kun se on jo käynnissä
- Montako pelaajaa samalla laitteella voi olla
- Mitä asetuksia pelaaja voi mukauttaa. Esim. pelaajan nimi.

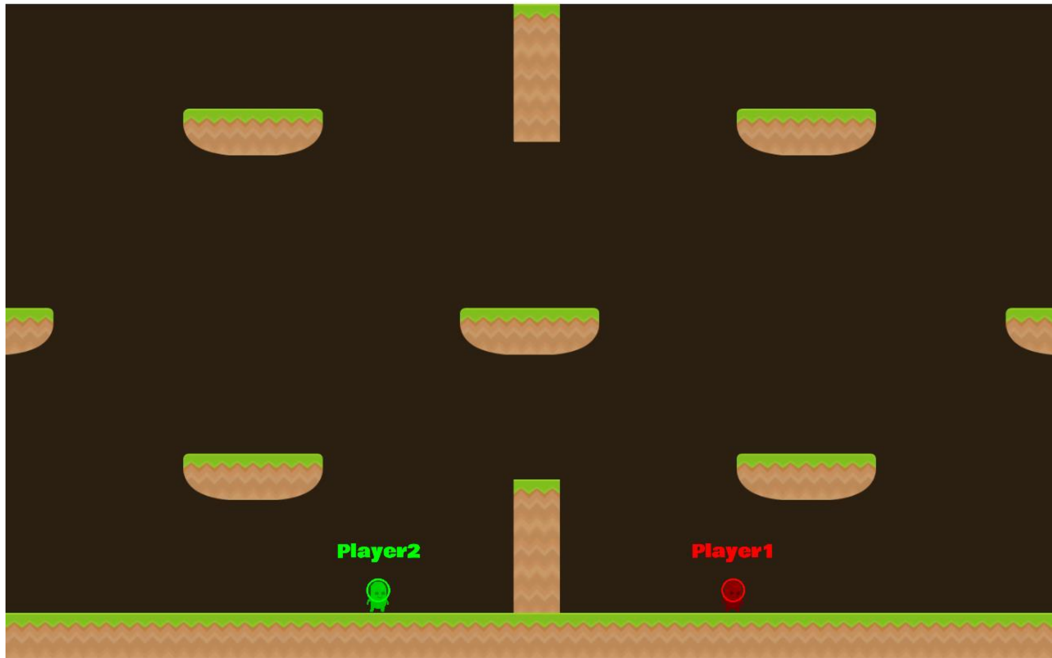
Tällä komponentilla on kahdenlaisia pelaajaobjekteja: LobbyPlayer ja GamePlayer. (Unity Manual)

4.4.2 Multiplayer Lobby Player ja Game Player

LobbyPlayer-objekti luodaan kun pelaaja liittyy palvelimelle tai samalle laitteelle lisätään uusi pelaaja. Se sisältää tiedon siitä onko pelaaja valmis ja mahdolliset pelaajakohtaiset asetukset pelaaja tekee aulassa.

GamePlayer-objekti luodaan kun varsinainen peli alkaa ja on se objekti, jota pelaaja ohjaa pelin aikana. Se tuhotaan kun peli on päättynyt ja pelaaja palaa takaisin aulaan. (Unity Manual)

5 PELIPROJEKTI



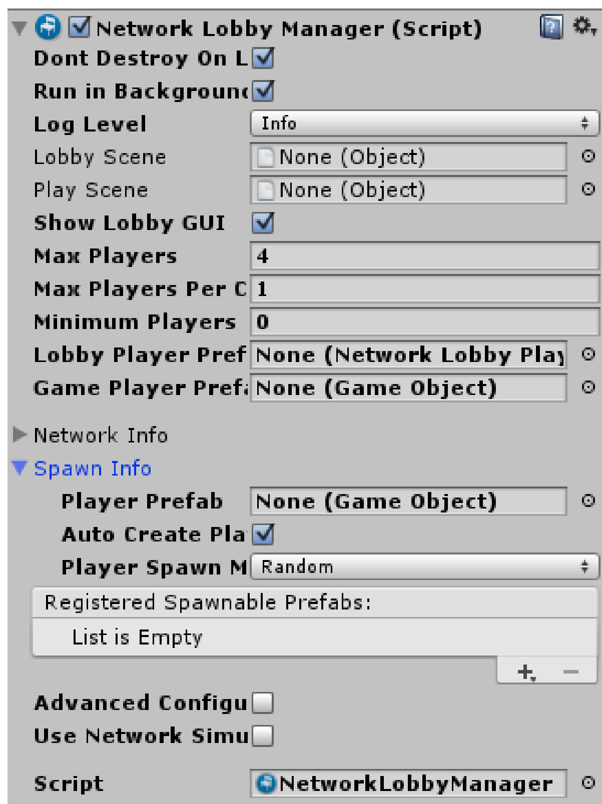
KUVA 12. Kuva pelitilanteesta.

Peliprojektin tavoitteena oli oppia käyttämään tässä työssä esiteltyä teoriaa käytännössä luomalla hyvin yksinkertainen, mutta toimiva prototyyppi. Toteutin projektin yksin ja siihen kului noin kolme viikkoa. Suurin osa tästä ajasta kului UNETin APIn opiskeluun ja tietojen testaamiseen käytännössä. Projektin tarkoitus ei ole olla kaupallinen peli, vaan se toimi eräänlaisena oppimisen työkaluna.

Projektin alussa yritin tehdä oman ratkaisun asiakkaan pelitilanteen ennustukseen ja latenssin hallintaan. Tämä osoittautui kuitenkin erittäin monimutkaiseksi ja aikaa vieväksi työksi ja lopulta oma ratkaisu ei tuottanut toivottua lopputulosta. Tämän vuoksi jouduin siirtymään käyttämään Unityn tarjoamaa ratkaisua, joka on rakennettu HHLAPI-järjestelmän sisään eikä vaadi erillistä koodia toimiakseen. Tätä opinnäytetyötä kirjoittaessa asiaan liittyvän teorian tunteminen on kuitenkin syventynyt ja oman ratkaisun toteuttaminen tuntuu paljon helpommalta kuin aikaisemmin.

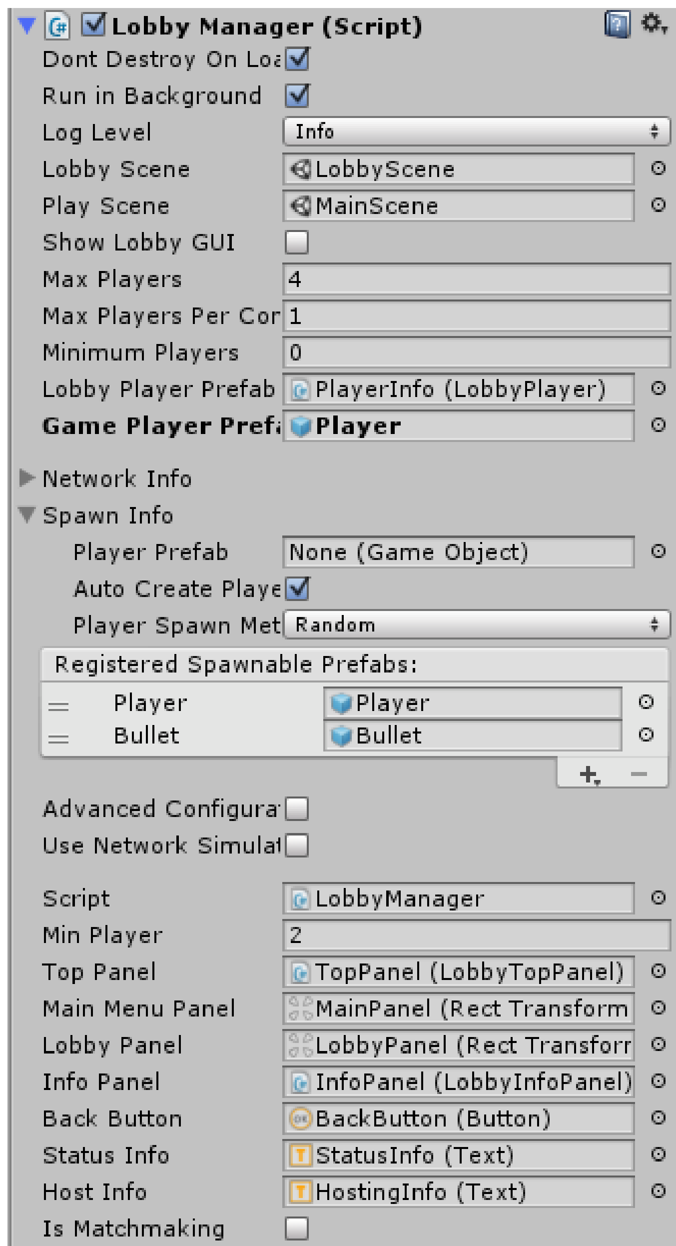
5.1 Aula

Projekti käyttää pohjanaan Asset Storesta ladattavaa Network Game Lobby (beta) -pakettia, joka tarjoaa yksinkertaisen drag&drop peliaularatkaisun. Tämä on rakennettu käyttäen Network Lobby Manager -komponenttia, mutta siihen on lisätty valmis UI. Ajan säästämiseksi valittiin tämä ratkaisu.



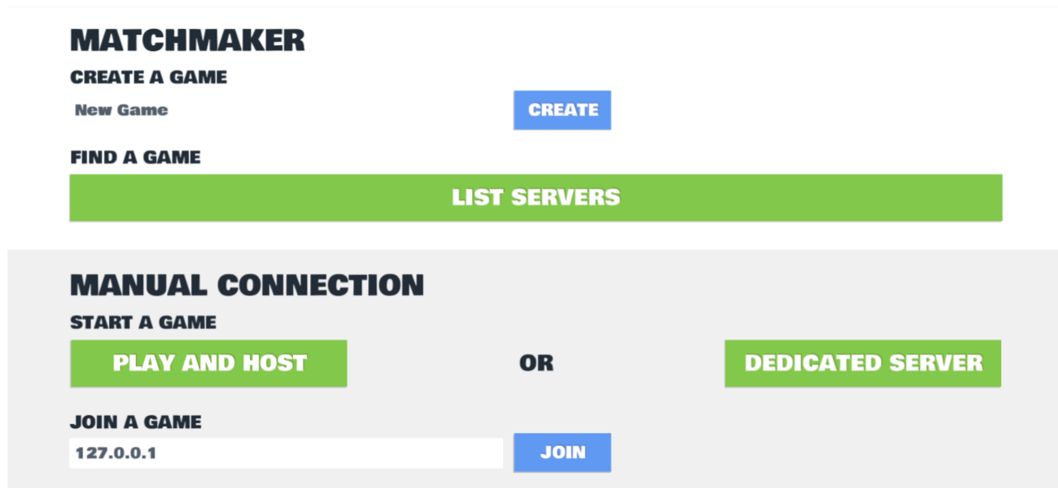
KUVA 13. NetworkLobbyManager-komponentti.

Kuvassa 13 on Unityn tarjoaman NetworkLobbyManager-komponentin perusversio. Se tarvitsee toimiakseen kaksi Scene-objektia: yhden aulalle ja toisen itse pelille. Scene-objekti eli näyttämö on Unityssa se alue, johon kaikki peliobjektit luodaan. Käytännössä se on siis pelialue. NetworkLobbyManager-komponentti tarvitsee myös objektin, joka sisältää Network Lobby Player –komponentin ja toisen objektin, joka kuvaa pelihahmoa itse pelissä, johon aulasta siirrytään. Jotta peliobjekteja voidaan luoda pelissä, ne täytyy lisätä Registered Spawnable Prefabs –listaan. Tämä komponentti mahdollistaa myös eritasoisten verkkojen simuloimisen testausta varten.



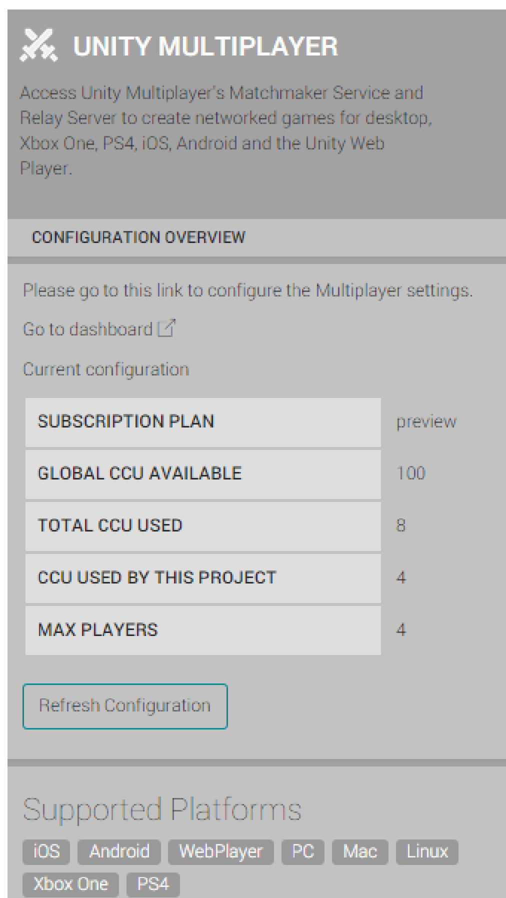
KUVA 14. Projektissa käytetty Lobby Manager.

Kuvan 14 Lobby Manager –komponentti on laajennettu versio kuvan 13 komponentista. Siihen on lisätty graafinen käyttöliittymä ja sen toiminta. Min Player –muuttuja määrittelee vähimmäismäärän pelaajia, jotta pelin pelaaminen onnistuu.



KUVA 15. Kuva peliaulasta

Aulasta on mahdollista luoda peli isäntänä (Play and Host), luoda pelkkä serveri (Dedicated Server) tai käyttää Unityn tarjoamaa Matchmaking-palvelua joko luomaan pelin tai etsimään valmiita pelejä verkosta. Matchmaking-palvelua käyttö vaatii Unity Multiplayer-konfiguraation tekemistä netissä. Koska Matchmaking-palvelu on vielä testivaiheessa, palvelu on ilmainen tietyin rajoituksin.



KUVA 16. Unity Multiplayer-palvelun konfiguraatio

5.2 Peli

Peli on 2-4 hengen 2D-peli, jossa pelaajat yrittävät ampua toiset pelaajat ja jäädä viimeiseksi henkiin. Peli loppuu kun joku pelaajista on voittanut 5 kierrosta. Tämän jälkeen kaikki pelaajat palaavat aulaan.

Pelin grafiikka koostuu spriteistä, eli pienistä kuvista ja kuvakokoelmista. Esimerkiksi pelaaja hahmossa on käytetty monta spriteä jotka yhdessä luovat sulavat animaatiot. Nämä spritet on lisenssoitu CC0 lisenssin alle. CC0 on yleisesti käytetty lisenssi public domain eli vapaastu yleiseen käyttöön asetetuille teoksille. Äänet jätettiin pelistä pois koska UNET ei vaikuta äänien lisäämiseen ja työ keskittyi juuri UNET-moninpelin tekemiseen.

5.2.1 Pelaajan liikkuminen

Pelaajan liikkuminen toteutettiin käyttäen Unityn omaa 2D-fysiikkamoottoria. Tämä mahdollisti painovoiman ja törmäyksentunnistuksen hyödyntämisen ilman erillistä koodia. Hyppimisen rajoittamiseksi pelaajan jalkojen alle tehdään törmäystunnistus käyttäen Physics2D.OverlapCircle() funktiota, joka palauttaa true-arvon, jos se löytää jonkin Collider2D komponenteista. Tätä rajoitettiin vielä niin, että törmäys tarkistetaan vain Ground-tasolta, jotta pelaaja ei voi hypätä toisen pelaajan tai ammuksen päältä.

```
    // Update is called once per frame
    private void Update ()
    {
        if (!isLocalPlayer)
            return;

        //Store the value of the input axis
        m_MovementInput = Input.GetAxis(m_MovementAxis);

    }

    private void FixedUpdate()
    {
        if (!isLocalPlayer)
            return;

        Move();
    }

    private void Move()
    {
        // Check to see if there is ground under the player
        grounded = Physics2D.OverlapCircle(groundCheck.position, groundRadius,
whatIsGround);

        // Update player velocity
        body.velocity = new Vector2(m_MovementInput * maxSpeed, body.velocity.y);

        // Check if we need to flip the sprite
        if (m_MovementInput > 0 && !facingRight)
        {
            Flip();
        }
        else if (m_MovementInput < 0 && facingRight)
        {
            Flip();
        }
        // If we are grounded and want to jump, jump
        if (grounded && Input.GetButtonDown("Jump"))
        {
            body.AddForce(new Vector2(0, jumpForce));
        }
    }
```

KUVA 17. Pelaajan liikkuminen koodissa.

5.2.2 Ampuminen

Pelaajalla ampumisesta tehtii hyvin yksinkertainen prototyyppiä varten. Pelaaja voi ampua niin kauan kun hänellä on ammuksia. Ammutun panoksen luomiseen käytettiin Command-kutsua nimeltä CmdFire(). Tämä funktio luo panoksen palvelimella, joka antaa tästä tiedon asiakkaille, jotta nämä luovat siitä oman versionsa.

```
    // Update is called once per frame
    [ClientCallback]
    private void Update ()
    {
        if (!isLocalPlayer)
            return;

        if(Input.GetButtonDown("Fire"))
        {
            if(m_Ammo > 0)
            {
                //Fire bullet
                Fire();
            }
        }
    }

    private void Fire()
    {
        m_Fired = true;
        Vector3 direction = Vector3.right * transform.localScale.x;
        CmdFire(m_Rigidbody2D.velocity, m_FireSpeed, direction, m_BarrelTransform.position,
Quaternion.identity);
        m_Ammo -= 1;
    }

    [Command]
    private void CmdFire(Vector3 rigidbodyVelocity, float launchForce, Vector3
shootingDirection, Vector3 position, Quaternion rotation)
    {
        //Instantiate the bullet and store a reference as Rigidbody2D
        Rigidbody2D bulletInstance = Instantiate(m_Bullet, position, rotation) as
Rigidbody2D;

        //Create a velocity for the bullet
        Vector3 velocity = rigidbodyVelocity + (launchForce * shootingDirection);
        velocity.y = 0;

        //Set the bullet's velocity to this velocity
        bulletInstance.velocity = velocity;

        NetworkServer.Spawn(bulletInstance.gameObject);

        if(transform.localScale.x < 0)
        {
            Vector3 theScale = bulletInstance.transform.localScale;
            theScale.x *= -1;
            bulletInstance.transform.localScale = theScale;
        }
    }
```

KUVA 18. Ammunta koodi.

5.2.3 Törmäyksen tunnistus

```
[ServerCallback]
private void OnTriggerEnter2D(Collider2D other)
{
    Rigidbody2D targetRigidbody2D = other.GetComponent<Rigidbody2D>();

    if(targetRigidbody2D)
    {
        //Check if the rigidbody2D has PlayerHealth component.
        PlayerHealth targetHealth = targetRigidbody2D.GetComponent<PlayerHealth>();

        if(targetHealth)
        {
            targetHealth.Damage(m_MaxDamage);
        }
    }
    if(!NetworkClient.active)
        PhysicForces();

    // Destroy the bullet on clients
    NetworkServer.Destroy(gameObject);
}

//called on client when the Network destroy that object (it was destroyed on server)
public override void OnNetworkDestroy()
{
    base.OnNetworkDestroy();
}

void PhysicForces()
{
    Rigidbody2D targetRigidbody2D = GetComponent<Rigidbody2D>();

    if(targetRigidbody2D ||
targetRigidbody2D.GetComponent<NetworkIdentity>().hasAuthority)
    {
        targetRigidbody2D.AddForce(Vector2.right * bulletSpeed);
    }
}
```

KUVA 19. Ammuksen törmäyksen tunnistus

Panoksen ja pelaajan törmäyksen tunnistus tapahtuu palvelimella. Tähän käytettiin Trigger-tyyppistä Collider-komponenttia. Trigger-asetus saa aikaan sen, että törmäys ei vaikuta fysiikkaan mitenkään, mutta se silti rekisteröidään. Panoksen osuessa pelaajaan tämän elämästä poistetaan panoksen aiheuttama vahinko ja panos tuhotaan. Tämä koodi suoritetaan vain palvelimella.

Käytännössä pelaaja kuolee yhdestä panoksesta sillä panoksen aiheuttama vahinko on enemmän kuin pelaajan elämä. Pelaajalle annettiin elämä ja panokselle vahinkoarvot jatkokehitystä ajatellen. Myöhemmin peliin voi lisätä esimerkiksi kilven, joka antaa pelaajan selvitä yhdestä panoksesta tuplaamalla tämän elämän.

6 POHDINTA

Nykyaikaiset pelimoottorit mahdollistavat pelien kehityksen hyvin pienellä joukolla. Yhden hengen tekemät hyvälaatuiset pelitkään eivät ole mahdottomuus. Tästä hyvänä esimerkkinä on kaupunginrakennuspeli Banished. Pelin kehittäminen vaatii edelleen paljon taitoa ja työtä, mutta työkaluja saatetaan amatöörien käsiin todella kovalla vauhdilla.

Unity on varsin helppokäyttöinen ja Unity Technologies panostaa paljon sen käytön opetukseen. Unityn omassa manuaalissa asiat on selitetty yksinkertaisesti ja kommentoitujen esimerkkien kanssa. Lisäksi he pitävät lähes viikottain Live Training stream -lähetyksen, jossa käydään kädestä pitäen läpi jokin Unityn ominaisuus tai projektin toteutus.

Uudet moninpelityökalut on suunniteltu hyvin ja niiden käytöstä on tehty erittäin yksinkertaista. Yksinkertaisimmissa tapauksissa koodia ei tarvitse kirjoittaa juuri ollenkaan. Itse en koskenut matalan tason APIin, mutta HLAPI tarjoaa hyvät työkalut suurimmalle osalle peleistä ja se automatisoi monia asioita, joita ei normaalisti edes huomaa, kuten pelaajan sijainnin interpolointia. Kaiken kaikkiaan Unity on onnistunut luomaan hyvän alustan niin pienille kuin suurillekin pelinkehittäjille.

LÄHTEET

Gaffer on Games. What every programmer needs to know about game networking, luettu 30.11.2015
http://gafferongames.com/networking-for-game-programmers/what-every-programmer-needs-to-know-about-game-networking/

Unity Asset Store. Network Game Lobby (beta), luettu 19.11.2015
https://www.assetstore.unity3d.com/en/#!/content/41836

Unityn blogi. Unity 5 Launch, luettu 20.11.2015
http://blogs.unity3d.com/2015/03/03/unity-5-launch/

Unityn blogi. Unity 5.1 is here!, luettu 20.11.2015
http://blogs.unity3d.com/2015/06/09/unity-5-1-is-here/

Unity Manual. Multiplayer and Networking, luettu 17.11.2015
http://docs.unity3d.com/Manual/UNet.html

Unity uutissivut. Unity Releases 2D Tools with 4.3 Update, luettu 20.11.2015
https://unity3d.com/company/public-relations/news/unity-releases-2d-tools-43-update

LIITTEET

Liite 1. Pelaajan liikkuminen

1(3)

```
using UnityEngine;
using UnityEngine.Networking;

public class PlayerMovement : NetworkBehaviour
{
    public int m_PlayerNumber = 1;          // Used to identify which player belongs to which
player.  This is set by this player's manager.
    public float m_Speed = 12f;             // How fast the player moves
    public Rigidbody2D m_Rigidbody2D;       // Reference used to move the character
    public Transform groundCheck;           // Location for the ground checking.
    public LayerMask whatIsGround;          // Used to dictate which layers are detected as
ground.
    private float groundRadius = 0.2f;      // How large the overlap circle for ground checking
is.


    private string m_MovementAxis;          // Name of the input axis for moving
    private float m_MovementInput;          // Current value of the movement input
    private float jumpForce = 700f;         // The force that is applied to the rigidbody when
jumping.
    private bool grounded = false;          // Is the player touching a ground. Used for jump
checking.
    private float maxSpeed = 6f;            // The maximum speed the player is allowed to move
at.
    private Rigidbody2D body;               // Reference to the Rigidbody2D component.

    [SerializeField]
    private bool facingRight;               // Current facing so we know when to flip the
sprite.

    private void Awake()
    {
        m_Rigidbody2D = GetComponent<Rigidbody2D>();
    }

    // Use this for initialization
    private void Start ()
    {
        m_MovementAxis = "Horizontal";
        body = GetComponent<Rigidbody2D>();
        facingRight = true;
    }


    // Update is called once per frame
    private void Update ()
    {
        if (!isLocalPlayer)
            return;

        //Store the value of the input axis
        m_MovementInput = Input.GetAxis(m_MovementAxis);

    }
```

2(3)

```
    private void FixedUpdate()
    {
        if (!isLocalPlayer)
            return;

        Move();
    }

    private void Move()
    {
        // Check to see if there is ground under the player
        grounded = Physics2D.OverlapCircle(groundCheck.position, groundRadius,
whatIsGround);

        // Update player velocity
        body.velocity = new Vector2(m_MovementInput * maxSpeed, body.velocity.y);

        // Check if we need to flip
        if (m_MovementInput > 0 && !facingRight)
        {
            Flip();
        }
        else if (m_MovementInput < 0 && facingRight)
        {
            Flip();
        }

        if (grounded && Input.GetButtonDown("Jump"))
        {
            body.AddForce(new Vector2(0, jumpForce));
        }
    }

    private void Flip()
    {
        Debug.Log("Flipped");
        //Flip code here
        facingRight = !facingRight;
        Vector3 theScale = transform.localScale;
        theScale.x *= -1;
        transform.localScale = theScale;
    }

    public void SetDefaults()
    {
        m_Rigidbody2D.velocity = Vector3.zero;
        m_Rigidbody2D.angularVelocity = 0f;
        m_Rigidbody2D.freezeRotation = true;
        m_Rigidbody2D.rotation = 0;



        m_MovementInput = 0f;
    }
```

3(3)

```
    //Freeze the rigidbody when the control is disabled to avoid drifting
    protected RigidbodyConstraints2D m_OriginalConstraints;
    void OnDisable()
    {
        m_OriginalConstraints = RigidbodyConstraints2D.FreezeRotation;
        m_Rigidbody2D.constraints = RigidbodyConstraints2D.FreezeAll;
    }

    void OnEnable()
    {
        m_Rigidbody2D.constraints = m_OriginalConstraints;
    }
}
```

Liite 2. Ampuminen


```
using UnityEngine;
using UnityEngine.UI;
using UnityEngine.Networking;

public class PlayerShooting : NetworkBehaviour
{
    public int m_PlayerNumber = 1;              // Used to identify the different players.
    public Rigidbody2D m_Bullet;                  // Prefab of the Bullet.
    public Transform m_BarrelTransform;         // A child of the character where the bullets
are spawned
    public AudioSource m_ShootingAudio;         // Reference to the audio source used to play
shooting sounds
    public AudioClip m_FireClip;                // Clip that plays when player fires


    private string m_FireButton;            // The input axis used for firing
    private Rigidbody2D m_Rigidbody2D;      // Reference to the rigidbody2D component
    private float m_FireSpeed;              // The speed the bullet moves at
    private bool m_Fired;                   // Whether or not the gun has been fired.
    private int m_Ammo;                     // How much ammo the player has


    private void Awake()
    {
        m_Rigidbody2D = GetComponent<Rigidbody2D>();
    }

    // Use this for initialization
    void Start ()
    {
        m_FireButton = "Fire";
        m_Ammo = 3;
        m_FireSpeed = 20f;
    }

    // Update is called once per frame
    [ClientCallback]
    private void Update ()
    {
        if (!isLocalPlayer)
            return;

        if(Input.GetButtonDown("Fire"))
        {
            if(m_Ammo > 0)
            {
                //Fire bullet
                Fire();
            }
        }
    }
```

2(2)

```
    private void Fire()
    {
        m_Fired = true;
        Vector3 direction = Vector3.right * transform.localScale.x;
        CmdFire(m_Rigidbody2D.velocity, m_FireSpeed, direction, m_BarrelTransform.position, 
Quaternion.identity);

        m_Ammo -= 1;

    }

    [Command]
    private void CmdFire(Vector3 rigidbodyVelocity, float launchForce, Vector3 
shootingDirection, Vector3 position, Quaternion rotation)
    {
        //Instantiate the bullet and store a reference as Rigidbody2D
        Rigidbody2D bulletInstance = Instantiate(m_Bullet, position, rotation) as 
Rigidbody2D;

        //Create a velocity for the bullet
        Vector3 velocity = rigidbodyVelocity + (launchForce * shootingDirection);

        velocity.y = 0;

        //Set the bullet's velocity to this velocity
        bulletInstance.velocity = velocity;

        NetworkServer.Spawn(bulletInstance.gameObject);

        if(transform.localScale.x < 0)
        {
            Vector3 theScale = bulletInstance.transform.localScale;
            theScale.x *= -1;
            bulletInstance.transform.localScale = theScale;
        }
    }
}
```

Liite 3. Ammuksen törmäyksen tunnistus

1(2)

```
using UnityEngine;
using System.Collections;
using UnityEngine.Networking;

public class BulletScript : NetworkBehaviour
{
    public ParticleSystem m_BulletParticles;        // Reference to the particle bullet trail
    public float m_MaxDamage;                       // The amount of damage done by the bullet
    public float m_MaxLifeTime = 2f;                // The amount of time the bullet stays
alive.
    public float bulletSpeed = 100f;                // The speed of the bullet


    private int m_PlayerMask;                       // Layer mask so only players are affected
by the bullets


    // Use this for initialization
    private void Start ()
    {
        if(isServer)
        {
            // Destroy bullet after lifetime is over.
            Destroy(gameObject, m_MaxLifeTime);
            GetComponent<Collider2D>().enabled = false;
            StartCoroutine(EnableCollision());
        }

        // Set the value of the layer mask.
        m_PlayerMask = LayerMask.GetMask("Players");
    }

    // Create a delay to avoid collider registering collision when spawned close to the
shooting player.
    IEnumerator EnableCollision()
    {
        yield return new WaitForSeconds(0.1f);
        GetComponent<Collider2D>().enabled = true;
    }

    // Triggers are handled only by the server
    [ServerCallback]
    private void OnTriggerEnter2D(Collider2D other)
    {
        Rigidbody2D targetRigidbody2D = other.GetComponent<Rigidbody2D>();

        //Check to see if the target has a Rigidbody2D component
        if(targetRigidbody2D)
        {

            //Check if the rigidbody2D has PlayerHealth component.
            PlayerHealth targetHealth = targetRigidbody2D.GetComponent<PlayerHealth>();

            if(targetHealth)
            {
                targetHealth.Damage(m_MaxDamage);
            }
        }
```

```
        if(!NetworkClient.active)
            PhysicForces();

        // Destroy the bullet on clients
        NetworkServer.Destroy(gameObject);

    }

    //called on client when the Network destroy that object (it was destroyed on server)
    public override void OnNetworkDestroy()
    {
        base.OnNetworkDestroy();
    }

    void PhysicForces()
    {
        //Find the rigidbody2D
        Rigidbody2D targetRigidbody2D = GetComponent<Rigidbody2D>();

        if(targetRigidbody2D ||
targetRigidbody2D.GetComponent<NetworkIdentity>().hasAuthority)
        {
            targetRigidbody2D.AddForce(Vector2.right * bulletSpeed);
        }
    }
}
```

Liite 4. Pelaajan elämän hallinta

1(2)

```
using UnityEngine;
using UnityEngine.UI;
using UnityEngine.Networking;

public class PlayerHealth : NetworkBehaviour {

    public float m_StartingHealth = 100f;             // The amount of health each player
starts with.
    public GameObject m_PlayerRenderers;              // References to all the Gameobjects that
need to be disabled when the player is dead.
    public PlayerSetup m_Setup;
    public PlayerManager m_Manager;                   //Associated manager, to disable control
when dying.


    [SyncVar(hook = "OnCurrentHealthChanged")]
    private float m_CurrentHealth;                    // How much health the player has.
    [SyncVar]
    private bool m_ZeroHealthHappened;                // Has the player been reduced to below 0
health?
    private BoxCollider2D m_Collider;                 // Disabled when the player is dead so it
doesn't collide with anything.


    private void Awake()
    {
        m_Collider = GetComponent<BoxCollider2D>();
    }

    //This is called when the player takes damage
    public void Damage(float amount)
    {
        //Reduce current health by the amount of damage done
        m_CurrentHealth -= amount;

        //If the current health is at or below zero and it has not yet been registered, call
OnZeroHealth
        if (m_CurrentHealth <= 0f && !m_ZeroHealthHappened)
        {
            OnZeroHealth();
        }
    }

    private void SetHealthUI()
    {
        //Show Health UI
    }

    void OnCurrentHealthChanged(float value)
    {
        m_CurrentHealth = value;
        // Change the UI elements appropriately.
        SetHealthUI();
    }
```

2(2)

```
    void OnZeroHealth()
    {
        // Set the flag so that this function is only called once
        m_ZeroHealthHappened = true;

        RpcOnZeroHealth();
    }

    private void InternalOnZeroHealth()
    {
        // Disable the collider and all the appropriate child gameobjects so the character
doesn't interact or show up when it's dead.
        SetPlayerActive(false);
    }

    [ClientRpc]
    private void RpcOnZeroHealth()
    {
        InternalOnZeroHealth();
    }

    private void SetPlayerActive(bool active)
    {
        m_Collider.enabled = active;

        m_PlayerRenderers.SetActive(active);
        //m_HealthCanvas.SetActive(active);

        if (active)
            m_Manager.EnableControl();
        else
            m_Manager.DisableControl();

        m_Setup.ActivateCrown(active);
    }

    public void SetDefaults()
    {
        m_CurrentHealth = m_StartingHealth;
        m_ZeroHealthHappened = false;
        SetPlayerActive(true);
    }
}
```