Tero Poutala

Pelin tallennus- ja latausjärjestelmän toteutus Unity-pelimoottorissa

Tradenomi

Tietojenkäsittely

Kevät 2020

KAMK • University of Applied Sciences

Tiivistelmä

Tekijä: Poutala Tero

Työn nimi: Pelin tallennus- ja latausjärjestelmän toteutus Unity-pelimoottorissa

Tutkintonimike: Tradenomi (AMK), tietojenkäsittely

Asiasanat: tallentaminen, lataaminen, Unity-pelimoottori, binäärijärjestelmän muoto, JSON, CONTINUE, Quest for Conquest

Tämä opinnäytetyö käsittelee tiedon tallentamisen eri muotoja Unity -pelimoottoria käyttäessä, antaa esimerkit binäärijärjestelmämuotoon ja JSON -muotoon tallentamisesta, tarkastelee kahden pelin, CONTINUE ja Quest for Conquestin, tallennus- ja latausjärjestelmiä, niiden eroja ja miten niitä olisi mahdollista parantaa.

Videopelit voivat tallentaa tietoa automaattisesti tai pelaajan antamalla komennolla. Automaattinen tallennus voidaan suorittaa uuteen kenttään siirtyessä ja pelaaja voi tallentaa videopelin tallennuspisteellä tai tallennusnäkymässä. Videopelin tiedot voidaan ladata, kun siirrytään kentästä toiseen. Tieto voidaan ladata huomaamattomasti pitkällä käytävällä tai hissillä. Latauksessa voidaan käyttää myös latausruutua, jossa näytetään latauksen progressio sekä liikkuva elementti. Videopelin tiedot voidaan tallentaa käyttäjälle paikallisesti tai tarvittaessa palvelimelle. Tieto voidaan tallentaa eri muotoon, hyödyntäen muodon ominaisuuksia.

Unity on kehitysalusta, joka tukee useaa eri järjestelmää sekä sisältää laajan valikoiman eri työkaluja. Unity tarjoaa eri tapoja oppia käyttämään kehitysalustaa opetusvideoiden, kurssien ja dokumentaation muodossa. Tietoa voidaan tallentaa binäärijärjestelmän muotoon luomalla tiedoston, tallentamalla tarvittavat tiedot luokkaan ja käyttämällä BinaryFormatter-objektia, muuttamalla ja kirjoittamalla luokan tiedot tiedostoon. Binäärijärjestelmämuotoista tietoa voidaan ladata etsimällä haluttu tiedosto, muuttamalla ja tallentamalla tieto, BinaryFormatter-objektia käyttäen, esimerkiksi luokkaan. Kun tietoa tallennetaan JSON-muotoon, luodaan tiedosto, muutetaan tieto string-muotoon ja kirjoitetaan se luotuun tiedostoon. JSON-muodosta ladattaessa etsitään haluttu tiedosto ja tallennetaan se, muutettuna, esimerkiksi luokkaan.

CONTINUE-pelissä tiedot tallennetaan, kun pelaaja aktivoi tallennuspisteen. Kenttään saapuessa haetaan kaikki tallennettavat objektit. Kun pelaaja siirtyy kentästä toiseen, voidaan joutua lataamaan kentän tiedot. Tiedot muutetaan käytettävään muotoon binäärijärjestelmämuodosta ja asetetaan haettuihin objekteihin. Quest for Conquest-pelissä tallennetaan esimerkiksi pelaajan ja pelaajan karavaanin tiedot JSON-muotoon matkatessa, taistelun jälkeen tai tapahtuman suoritettua. Tietojen tallennuksessa ja latauksessa hyödynnetään videopelin eri manageреja.

Minkälainen tallennus- ja latausjärjestelmä sopii videopeliin, riippuu videopelin ominaisuuksista. Yksinpelin järjestelmän voi toteuttaa käyttämällä JSON -muotoa ja tallentaa paikallisesti. CONTINUE- ja Quest for Conquest -videopelien järjestelmien suurimpia eroja ovat eri muotoihin tallentaminen ja CONTINUE:n useampi tallennustiedosto. Molemmat järjestelmät tarvitsisivat optimointia, sekä CONTINUE:n useampi tiedosto voisi muuttaa yhdeksi tiedostoksi. Molempien videopelien järjestelmät voisi myös muuttaa tukemaan useampaa samanaikaista tallennusta.

Opinnäytetyötä varten on tehty kaksi esimerkkijärjestelmää, joista toinen on binäärijärjestelmämuodossa ja toinen JSON-muodossa. Esimerkkien selitystä tukemassa käytettiin Microsoftin ja Unityn omaa dokumentaatiota aiheesta. CONTINUE- ja Quest for Conquestin järjestelmien kertomisen pohjalla on käytetty pelien lähdekoodia.

Abstract

Author: Poutala Tero

Title of the Publication: Creating a Saving and Loading Systems in Unity Game Engine

Degree Title: Bachelor of Business Administration, Information Technology

Keywords: saving, loading, Unity game engine, binary, JSON, CONTINUE, Quest for Conquest

This thesis examines different ways of saving and loading data in general and while using Unity game engine. The thesis will give examples of saving and loading data using binary and JSON. After the examples, the thesis takes a closer look at CONTINUE's and Quest for Conquest's saving and loading systems, how these two systems differ and how the systems could be improved.

Video games can save data automatically or by the player's command. Saving automatically can be executed on transforming to a new level. The player can save the game on checkpoint or in a saving view. The game might load data on moving transitioning between levels. The data can be loaded seamlessly with a long hallway or with an elevator. A loading screen can be used for loading, as well. If the loading screen is used, there should be a progress bar and some moving element on the screen. A video game's data can also be saved in a server. The data can be saved in different formats and the different formats' properties can be utilized.

Unity is a development platform which supports different systems and includes a wide selection of tools. Unity offers different ways to learn about the development platforms. These include tutorial videos, courses and documentation. Data can be saved into a binary format by creating a file, saving all the necessary data into a class and serializing and writing the data stored in the class to the created file by using BinaryFormatter object. If data is saved using JSON format, we create a file, serialize the data into a string format and write the data into the file. When we need to load JSON format data, we need to find the file, deserialize the data and set the data into a class, for example.

The data is saved in CONTINUE the video game when player activates a checkpoint. We find all the objects which need saving on transitioning to a level. We might need to load the level's data when the player is moving between two levels. The data is deserialized from binary format and set to the found objects. We save the player's values and caravan values on saving the game in Quest for Conquest. The data is saved to JSON format. Saving the game can be triggered on traveling, after a combat or on completing an event. Quest for Conquest utilizes managers on saving and loading data.

Which kind of saving and loading system a video game needs is up to the video game's properties. A single player game can be saved locally by using JSON format. The biggest differences between CONTINUE's and Quest for Conquest's systems are different save formats and CONTINUE's multiple save files. Both systems need optimization and CONTINUE's system's multiple files should be changed to a single save file. Both systems could be upgraded to support multiple simultaneous save files.

One binary format and one JSON format saving and loading system was created for an example to this thesis. Examples are explained using Unity's and Microsoft documentation about the topic. We used the source code of both CONTINUE and Quest for Continue to explain how saving and loading systems work on these video games.

Sisällys

1 Johdanto 1

2 Tallentaminen ja lataaminen videopeleissä 2

3 Unity-pelimoottori, tietojen tallentaminen ja lataaminen 5

3.1 Esimerkki binäärijärjestelmämuotoon tallentamisesta ja sen lataamisesta 6

3.2 Esimerkki JSON:iin tallentamisesta ja lataamisesta 9

4 Tiedon tallentaminen ja lataaminen CONTINUE-pelissä 12

5 Tiedon tallentaminen ja lataaminen Quest for Conquest-pelissä 15

6 CONTINUE ja Quest for Conquestin järjestelmien vertailua 17

7 Miten parantaisin CONTINUE:n ja Quest for Conquestin järjestelmiä 18

8 Yhteenveto 19

Lähteet 20

Käsiteluettelo

Binäärijärjestelmämuoto: Opinnäytetyössä binäärijärjestelmämuoto tarkoittaa biteiksi muutettua objektia, jota käytetään tallentamisessa ja lataamisessa.

CONTINUE: CONTINUE on nopeatempoinen toimintapeli Windows-alustalle.

Deserialization: Deserialization tarkoittaa tavujonon muuttamista takaisin alkuperäiseksi kohteeksi, jotta tallennettua tietoa voidaan hyödyntää.

JSON: JSON on muoto, mihin objekti voidaan muuttaa käyttämällä Unityn JsonUtility-luokkaa.

Lataaminen: Tietoa ladataan käyttöä varten tallennetusta kohteesta. Esim. videopeli lataa tiedot ja asettaa ne ennen pelin alkua tietokoneen kovalevylle tallennetusta tiedostosta.

Quest for Conquest: Quest for Conquest on seikkailupeli, joka on julkaistu Windows-alustalle.

Serialization: Serialization tarkoittaa kohteen muuttamista tavujonoksi tallentamista varten, jotta tietoa voidaan hyödyntää myöhemmin.

Tallentaminen: Tietoa tallennetaan myöhempää käyttöä varten. Esim. videopelin tiedot tallennetaan tallennuspisteellä tietokoneen kovalevylle.

1 Johdanto

Opinnäytetyö on tehty kertomaan tallennus- ja latausjärjestelmistä, miten ne ilmenevät videopeleissä ja miten eri tavoin videopelin tietoa voidaan tallentaa. Opinnäytetyössä annetaan yksinkertaiset esimerkit binäärijärjestelmämuotoisesta ja JSON-muotoisesta tallennus- ja latausjärjestelmästä, sekä käydään läpi, miten järjestelmät on toteutettu kahdessa eri pelissä, CONTINUE ja Quest for Conquest. Järjestelmiä myös vertaillaan keskenään ja käsitellään, miten järjestelmiä voisi parantaa.

Opinnäytetyön aluksi käydään läpi lyhyesti tallennus- ja latausjärjestelmistä ja mihin eri muotoihin tietoa voi tallentaa. Kun tämä on käyty läpi, selitetään tallennus- ja latausvaihtoehdoista Unity-pelimoottoria käyttäessä. Myös binäärijärjestelmämuodosta ja JSON-muodosta annetaan esimerkit. Esimerkit sisältävät myös kuvia valmiista järjestelmistä. Kun esimerkit on käyty läpi, aletaan tarkastelemaan CONTINUE- ja Quest for Conquest-pelien tallennus- ja latausjärjestelmiä. Järjestelmien toimintaa käydään läpi, vertaillaan järjestelmien eroja, sekä lopuksi suunnitellaan, miten järjestelmiä voisi parantaa.

2 Tallentaminen ja lataaminen videopeleissä

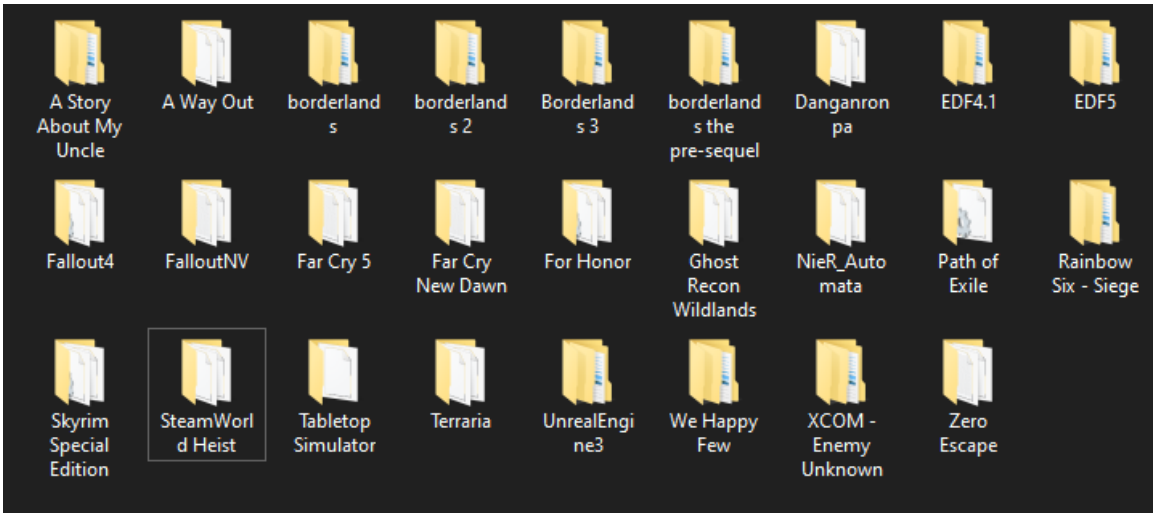
Videopeleissä tietoa tallennetaan, jotta pelaajan ei tarvitse aloittaa peliä joka käynnistyskerralla alusta tai asettaa asetuksia (esim. ohjain- ja ääniasetukset) uudelleen. Videopelien tallentaminen voidaan suorittaa automaattisesti, ilman minkäänlaista komentoa pelaajalta. Tällaista tallennusta kutsutaan automaattitallennukseksi. Automaattitallennukset ovat hyödyllisiä, jos pelaaja ei muista tallentaa videopeliä pelatessaan tai jos videopeli yllättäen sulkeutuu virheen vuoksi. Automaattitallennuksia voidaan tehdä, kun videopeli on ladannut uuden alueen. Videopelin voi tallentaa myös pelaajan komennolla. Pelaaja voi aktivoida tallentamisen käymällä tallennuspisteellä tai tekemällä tallennuksen videopelin taukovalikossa. Tallennuspisteet ovat videopelin kenttään asetettuja paikkoja, jotka voivat olla pelaajalle näkyviä tai näkymättömiä. Tallentamisen aktivointi voi vaatia pelaajalta komentoa tallennuspisteellä, mutta tallentaminen on hyvä suorittaa automaattisesti. [1, s. 192, 345, 346, 363.] Pelaaja voi myös tallentaa pelin manuaalisesti tallennusnäkymässä. Tallennusnäkymään voi päästä taukovalikon kautta. Kuvassa 1 on pelin Viscera Cleanup Detail tallennusnäkymä. Pelaaja voi luoda uuden tallennuksen painamalla New Saved Game -nappia. Pelaaja voi tallentaa myös jo olemassa olevaan tallennukseen. Tämä kirjoittaa vanhan tallennuksen tietojen päälle nykyisen pelin tilan. Kun pelaajan progressio tai pelin asetukset on tallennettu paikallisesti, tiedot on helppo ladata ja asettaa ohjelman käynnistyessä ja pelaaja pääsee jatkamaan suoraan siitä, mihin viime kerralla jäi.

Kuva 1. Viscera Cleanup Detail -pelin tallennusikkuna.

Videopeli alkaa lataamaan tietoja esimerkiksi, kun pelaaja on siirtymässä kentästä toiseen. Tällaisessa tilanteessa peli lataa seuraavan kentän tiedot, mikä voi viedä aikaa. Lataus voidaan suorittaa huomaamattomasti tai latausruudulla. Jos videopelin lataus suoritetaan huomaamattomasti, pelaaja voi joutua odottamaan oven avautumista tai hissin saapumista oikeaan kerrokseen. Jos lataus suoritetaan latausruutua käyttäen, on pelaajalle hyvä kertoa, miten paljon peli on ladannut. Tämä voidaan toteuttaa latauspalkilla tai prosenttiluvulla. Latausruudussa on myös hyvä olla jokin liikkuva elementti tai hahmo, jota pelaaja voi käännellä ja liikutella. Tämä voi kertoo pelaajalle, että pelin latauksen aikana ei ole tapahtunut virhettä. [1, s. 192, 193.]

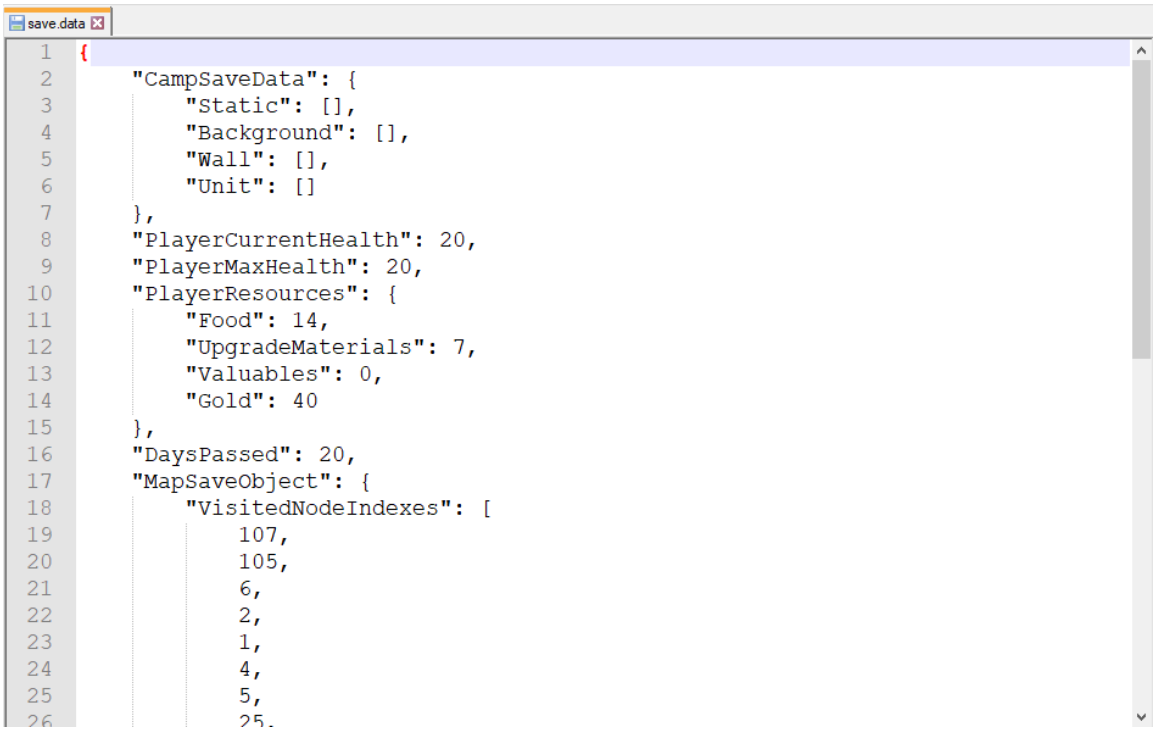
Pelin ominaisuuksien mukaan pelin tietoa voidaan tallentaa eri tavoin. Jos peli on yksin pelattava seikkailupeli, tiedon tallennus voidaan tehdä käyttäjälle paikallisesti. Kuvassa 2 on tietokoneelta löytyvä My Games -kansio, mihin useat pelit tallentavat tietoja. Tarvittaessa paikalliset tiedot on mahdollista tallentaa palvelimelle, mitä ominaisuutta esim. Steam-sovellus tukee.

Steam-sovelluksen Steam Cloud -ominaisuus antaa käyttäjälle mahdollisuuden tallentaa pelin tallennettavat tiedostot Steamin palvelimelle. Kun pelaaja on sulkenut pelin, Steam Cloud API:a käyttäen voi luoda, muokata ja poistaa tiedostoja Steamin palvelimelta. Tämä myös mahdollistaa palvelimelle tallennettujen tietojen synkronoinnin usean eri laitteen välillä, jos pelaaja käyttää Steamia usealla eri laitteella. [2.]



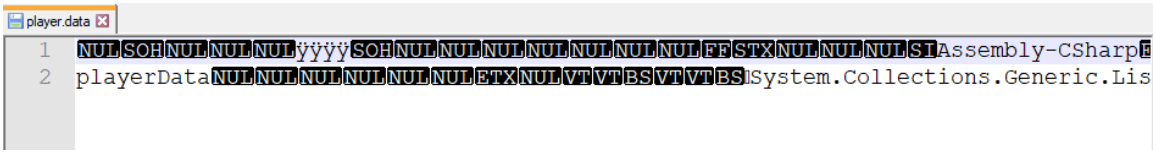
Kuva 2. My Games -kansio sisältää useiden eri pelien tallennuksia ja tallennettuja asetuksia

Tietoa voidaan tallentaa tiedostoon eri muodoissa. Esimerkkejä näistä muodoista ovat JSON ja binäärijärjestelmän muoto. JSON muodossa teksti tallennetaan muokattavaan ja luettavaan muotoon, mikä mahdollistaa kehitysvaiheessa helpon tarkistamisen virheiden varalta (kuva 3). Quest for Conquest -peli käyttää JSON -muotoa tallennustiedostoissaan.



Kuva 3. JSON muotoon tallennettua tietoa

Binäärijärjestelmän muoto tallentaa tiedon tavujonoksi, mitä on vaikea lukea ja muokata (kuva 4). Tämä antaa mahdollisuuden tallentaa tietoa, jota ei haluta pelaajan muokkaavan. CONTINUE-peli käyttää binäärijärjestelmämuotoa tallennustiedostoissaan.



Kuva 4. Binäärijärjestelmän muotoon tallennettua tietoa

Tulevissa luvuissa käsitellään, miten tallentamis- ja lataamisjärjestelmän voi luoda Unity-pelimoottoria käyttäessä ja myöhemmin verrataan kahden eri pelin, CONTINUE:n ja Quest for Conquestin, tallennus- ja latausjärjestelmiä, miten ne toimivat, miten ne eroavat toisistaan, sekä miten näitä järjestelmiä olisi mahdollista parantaa.

3 Unity-pelimoottori, tietojen tallentaminen ja lataaminen

Unity on kehitysalusta, joka tarjoaa useammalle alalle työkaluja. Yksi näistä aloista on videopelit, joihin myös tämä opinnäytetyö keskittyy. Esimerkkeinä Unityllä tuotetuista peleistä ovat mm. taistelupeli Escape from Tarkov sekä seikkailupelit Hollow Knight ja Ghost of a Tale [3]. Unityllä on mahdollista tuottaa videopelejä eri alustoille. Unity tukee kehittämistä eri tietokonejärjestelmille (Windows, Mac ja Linux), konsoleille (PS4, Xbox One ja Nintendo Switch), puhelinjärjestelmille (Android ja iOS) sekä virtuaalitodellisuutta [4]. Unityn tarjoamia työkaluja sovelluksien tuottamiseen ovat mm. grafiikan ja fysiikan hallinta ja monetisaatio [5]. Työkalujen lisäksi Unity tarjoaa usean eri tavan oppia käyttämään kehitysalustaa. Unityä ja sen eri ominaisuuksia voi oppia käyttämään katsomalla opetusvideoita, käymällä läpi eri kursseja tai lataamalla ja muokkaamalla itse ilmaisia valmiita projekteja [6]. Unity myös sisältää dokumentaatiota, jota voi käyttää sovellusta käyttäessä [7]. Tämän vuoksi Unity sopii kaikille kehittäjille taitotasosta riippumatta. Unity tarjoaa eritasoisille kehittäjille ja yrityksille erilaisia lisenssejä.

Unity tarjoaa Student-lisenssin opiskelijoille ja yksityishenkilöille Personal-lisenssin, mikä tekee kehitysalustasta vielä helpommin lähestyttävän aloittelijoilla. Lisenssit kuitenkin sisältävät vaatimuksen, joka on täytyttävä. Opiskelijalisenssin hakijan on oltava osana GitHub Student Developer Pack -ohjelmaa, sekä Personal-lisenssin haltijan ei saa tuottaa yli $100 000 viimeisen 12 kuukauden aikana Unityllä kehitetyllä sovelluksella. Jos Personal-lisenssin omistaja tienaa enemmän kuin $100 000 vuodessa, on siirryttävä maksullisiin lisensseihin. Unityn maksulliset lisenssit ovat Plus- ja Pro-lisenssit. Nämä lisenssit tuovat mukanaan myös uusia ominaisuuksia, kuten Plus-lisenssin avaaman pääsyn maksulliseen opiskelumateriaaliin. [8.]

Tietoa tallentaessa kehittäjä voi käyttää Unityn omia metodeja, kuten ToJson ja FromJson, JSON-muotoon tallentaessa [9, 10]. Muihin muotoihin, kuten binäärijärjestelmän muotoon, C#-ohjelmointikieli tarjoaa oman ratkaisun [11]. Tallennus- ja latausjärjestelmää suunnitellessa on myös hyvä ottaa huomioon Unityn tarjoamat ScriptableObject- ja PlayerPrefs-ominaisuudet. ScriptableObjectia voi käyttää tietojen säilyttämiseen ja myöhemmin tallentamiseen, sekä siihen lataamiseen [12]. PlayerPrefsiä voi käyttää esim. pelaajan asettamien asetusten tallentamiseen [13].

3.1 Esimerkki binäärijärjestelmämuotoon tallentamisesta ja sen lataamisesta

Tässä luvussa annetaan yksinkertainen esimerkki tallennus- ja latausjärjestelmästä binäärijärjestelmämuotoa käyttäen. Esimerkissä käytetään BinaryData-nimistä luokkaa, johon asetetaan tiedot ja muutetaan myöhemmin tallennettavaan muotoon (kuva 6). Esimerkissä käytetään asetettuja tiedostopolkuja, -nimiä ja -päätteitä (kuva 5).

```
private string _UnityPath = "";
private string _MyGamesPath = "";
private string _FileNameBinary = "/Save.bin";
private string _FileNameJson = "/Save.json";

0 references
private void Awake()
{
    _UnityPath = Application.persistentDataPath;
    _MyGamesPath = Environment.GetFolderPath(Environment.SpecialFolder.MyDocuments) + "/My Games/MyGame";
}
```

Kuva 5. Esimerkissä käytettävät tiedostopolut, -nimet ja -päätteet

```
[Serializable] public class BinaryData
{
    private string _Name = "";
    private int _Score = 0;
    private float _Timer = 0f;

    2 references
    public BinaryData(string name, int score, float timer)
    {
        _Name = name;
        _Score = score;
        _Timer = timer;
    }

    0 references
    public void DebugData()
    {
        Debug.LogFormat("Name: {0}, Score: {1}, Timer {2}", _Name, _Score, _Timer);
    }
}
```

Kuva 6. BinaryData-luokka

BinaryData-luokka sisältää _Name-, _Score- ja _Timer-muuttujat, joihin asetetaan tallennettavat arvot. Luokka sisältää myös rakentaja- ja DebugData-metodit. Rakentajametodilla asetetaan arvot luokan muuttujille luokkaa luodessa. DebugData-metodilla kirjoitetaan Unityn konsoliin pieni viesti, joka kertoo luokan muuttujien arvot. DebugData-metodi on tarkoitettu vain kehitysvaiheessa tietojen tarkistamiseen, ja myöhemmin sen voi poistaa.

```
private void SaveBinary(string path)
{
    if (!Directory.Exists(path))
        Directory.CreateDirectory(path);

    FileStream file = new FileStream(path + _FileNameBinary, FileMode.OpenOrCreate);
    BinaryData data = new BinaryData("Adam", 100, 60f);
    BinaryFormatter binaryFormatter = new BinaryFormatter();

    try
    {
        binaryFormatter.Serialize(file, data);
    }
    catch (SerializationException e)
    {
        Debug.Log("Couldn't serialize: " + e);
        throw;
    }
    finally
    {
        file.Close();
    }
}
```

Kuva 7. SaveBinary-metodi, jolla tallennetaan tietoa binäärijärjestelmämuotoon

Kun tiedot halutaan tallentaa, kutsutaan SaveBinary-metodia (kuva 7). Metodi alkaa tarkistuksella, löytyykö annettu tiedostopolku tietokoneelta. Jos tiedostopolkua ei löydetä, se luodaan. Kun tarkistus on tehty, luodaan FileStream-tyyppinen objekti. Tämän objektin luomiseen käytetään tarkistettua tiedostopolkua, asetettua tiedostonimeä ja -päätettä sekä FileMode-parametria [14]. Tiedostopolku sekä haluttu tiedostonimi ja -pääte annetaan objektia luodessa, koska tällä osoitetaan, mihin tallennustiedosto tulee luoda [14]. FileMode.OpenOrCreate-parametri kertoo luotavalle objektille, että avaa tai luo tiedosto [15]. Jos tiedosto on jo olemassa, uuden tiedoston luomisen sijaan, vanha tiedosto avataan [15]. Kun tiedosto on käsitelty, seuraavaksi luodaan luokka, johon tiedot tallennetaan. Luotava luokka on BinaryData-tyyppiä ja luokalle voidaan tässä tilanteessa antaa mitkä arvot tahansa, kunhan annettavat parametrit ovat oikeaa muotoa. Esimerkissä annetaan _Name-muuttujalle Adam, _Score-muuttujalle 100 ja _Timer-muuttujalle 60f-arvot. Kun BinaryData-luokka on luotu, seuraavaksi luodaan BinaryFormatter-objekti. Tämän objektin avulla luokka voidaan muuttaa binäärijärjestelmämuotoon Serialize-metodilla [16, 17]. Uutta saman tyypin objektia käytetään myöhemmin myös tietoa ladatessa. Seuraavaksi yritetään muuttaa luotu BinaryData-luokka binäärijärjestelmämuotoon annettuun tiedostoon. Serialize-metodille annetaan file- ja data-muuttujat [17]. File-muuttuja sisältää tiedostopolun, tiedoston nimen sekä tiedostopäätteen, mihin tieto tullaan tallentamaan ja data-muuttuja sisältää tallennettavan tiedon. Jos muunnoksen aikana tapahtuu virhe, virhe kirjoitetaan Unityn konsoliin ja

Serialize-metodi jätetään kesken. Jos muutos onnistui ilman virheitä, lopuksi suljetaan tiedosto, johon tieto tallennettiin. Tiedosto suljetaan, jotta sitä voidaan käyttää myöhemmin. Jos tiedosto jätetään sulkematta, sitä ei voida kirjoittaa tai lukea.

```
private void LoadBinary(string path)
{
    if (Directory.Exists(path) && File.Exists(path + _FileNameBinary))
    {
        BinaryData data = new BinaryData("", 0, 0f);
        FileStream file = new FileStream(path + _FileNameBinary, FileMode.Open);
        BinaryFormatter binaryFormatter = new BinaryFormatter();

        try
        {
            data = (BinaryData)binaryFormatter.Deserialize(file);
        }
        catch (SerializationException e)
        {
            Debug.Log("Couldn't deserialize: " + e);
            throw;
        }
        finally
        {
            file.Close();
        }
    }
    else
    {
        Debug.Log("Directory or File didn't exist. Can't finish loading data.");
    }
}
```

Kuva 8. Tiedon lataamisessa käytettävä LoadBinary -metodi ja sen rakenne.

Kun tiedosto on ladattava, käytetään LoadBinary-metodia (kuva 8). Metodin aluksi tarkistetaan, onko haluttu tiedostopolku ja tiedosto olemassa tietokoneella. Jos tiedostopolkua tai tiedostoa ei ole, annetaan lataamisen sijasta Unityn konsoliin viesti. Viestiä ei ole pakko kirjoittaa, ja sen voi halutessaan poistaa, kun järjestelmä on testattu toimivaksi. Jos tarkistuksesta päästään läpi, luodaan luokka, johon ladattava tieto asetetaan. Luokkaa ei tarvitse luoda, jos on jo olemassa kohde, mihin tieto ladataan, mutta tässä esimerkissä kohdetta ei ole. Luokan arvoilla ei ole tässä vaiheessa väliä, koska asetetut arvot korvataan ladatuilla arvoilla. Luokan luonnin jälkeen avataan tallennustiedosto luomalla uusi FileStream-objekti ja antamalla tälle tiedostopolku, -nimi ja -pääte sekä FileMode.Open [14]. FileMode.Open antaa metodin luodessa tiedon, että tiedosto avataan [15]. Myös uusi BinaryFormatter-objekti luodaan Deserializa-metodin käyttöä varten [16, 18]. Kun objektit on luotu, seuraavaksi yritetään muuttaa tallennustiedoston tieto luokan muo-

toon. Ensiksi yritetään muuntaa tallennettu binäärijärjestelmän muoto annettuun luokkaan antamalla BinaryFormatter-objektille tieto, minkä luokan muotoon tieto halutaan muuttaa, sekä tiedosto, missä muutettava tieto on [16]. Tässä esimerkissä tieto muutetaan BinaryData-luokan muotoon. Jos prosessin aikana tapahtuu virhe, kirjoitetaan Unityn konsoliin pieni viesti ja lopetetaan prosessi. Jos mitään virheitä ei tapahdu, lopuksi suljetaan tiedosto, josta tieto muutettiin. Kuvassa 9 on kirjoitettu tiedostosta ladattu tieto Unityn konsoliin luettavaksi.

Kuva 9. Tiedostosta ladattu tieto on kirjoitettu Unityn konsoliin BinaryData-luokan DebugData-metodin avulla.

3.2 Esimerkki JSON:iin tallentamisesta ja lataamisesta

Tässä luvussa annetaan yksinkertainen esimerkki tallennus- ja latausjärjestelmästä JSON:ia käyttäen. Tässä esimerkissä käytetään JsonData nimistä luokkaa, jotka edellisen binäärijärjestelmämuoto esimerkin mukaisesti sisältää samat muuttujat (Name-, Score- ja Timer-muuttujat), mutta muuttujat ovat public-näkyvyysmäärettä private-näkyvyysmääreen sijasta (kuva 10). Jos halutaan käyttää private-näkyvyysmäärettä, on asetettava muuttujan eteen SerializableField-attribuutti [19]. Tämä luokka sisältää myös rakentaja ja DebugData-metodit ja ne toimivat samoin, kuin binäärijärjestelmämuodon esimerkin luokassa. Esimerkki käyttää myös samoja tiedostopolkuja, -nimiä ja -päätteitä, kuin aikaisempi esimerkki.

```
[Serializable] public class JsonData
{
    public string Name = "";
    public int Score = 0;
    public float Timer = 0f;

    1 reference
    public JsonData(string name, int score, float timer)
    {
        Name = name;
        Score = score;
        Timer = timer;
    }

    1 reference
    public void DebugData()
    {
        Debug.LogFormat("Name: {0}, Score: {1}, Timer {2}", Name, Score, Timer);
    }
}
```

Kuva 10. JsonData-luokan arkenne

```
private void SaveJson(string path)
{
    if (!Directory.Exists(path))
    {
        Directory.CreateDirectory(path);
    }

    if (!File.Exists(path + _FileNameJson))
    {
        FileStream file = new FileStream(path + _FileNameJson, FileMode.Create);
        file.Close();
    }

    JsonData json = new JsonData("Kate", 150, 210f);
    string data = JsonUtility.ToJson(json);
    File.WriteAllText(path + _FileNameJson, data);
}
```

Kuva 11. Tallentamiseen käytettävä SaveJson-metodi ja sen rakenne

Kun tietoa tallennetaan, kutsutaan SaveJson-metodia (kuva 11). Metodin aluksi tarkistetaan, onko metodille annettu tiedostopolku olemassa. Samaan tapaan, kuin binäärijärjestelmämuodon esimerkissä, jos tiedostopolkua ei ole, se luodaan. Kun tiedostopolku on luotu, tarkistetaan jos tallennustiedosto on jo olemassa. Jos tiedostoa ei ole jo olemassa, se luodaan antamalla luotavalle FileStream-tyyppiselle objektille tiedostopolku, -nimi, -pääte, sekä FileMode.Create [14]. Tilanteessa käytetään FileMode.Create, koska tiedetään, että tiedostoa ei ole jo olemassa. Luonnin jälkeen suljetaan tiedosto, jotta sitä voidaan käyttää myöhemmin. Kun tiedosto on luotu, luodaan JsonData-tyyppinen luokka. Tähän luokkaan tallennetaan halutut arvot. Esimerkissä luokalle asetetaan seuraavat arvot sitä luodessa: Kate, 150 ja 210f. Kun luokka on luotu ja tiedot asetettu, muutetaan se string-muotoon käyttämällä JsonUtility.ToJson-metodia. Tämä metodi muuttaa

luokan kirjainjonoksi, jonka jälkeen tätä käytetään tiedostoon kirjoittamiseen [9]. Tähän käytetään File.WriteAllText-metodia. Metodille syötetään tiedostopolku, -nimi, -pääte ja kirjoitettava teksti [20]. File.WriteAllText-metodi avaa tiedoston ja sulkee sen automaattisesti, joten sen itse tekemisestä ei tarvitse huolehtia [20].

```
private void LoadJson(string path)
{
    if (Directory.Exists(path) && File.Exists(path + _FileNameJson))
    {
        string loadedData = File.ReadAllText(path + _FileNameJson);
        JsonData data = JsonUtility.FromJson<JsonData>(loadedData);
    }
    else
    {
        Debug.Log("Directory or File didn't exist. Can't finish loading data.");
    }
}
```

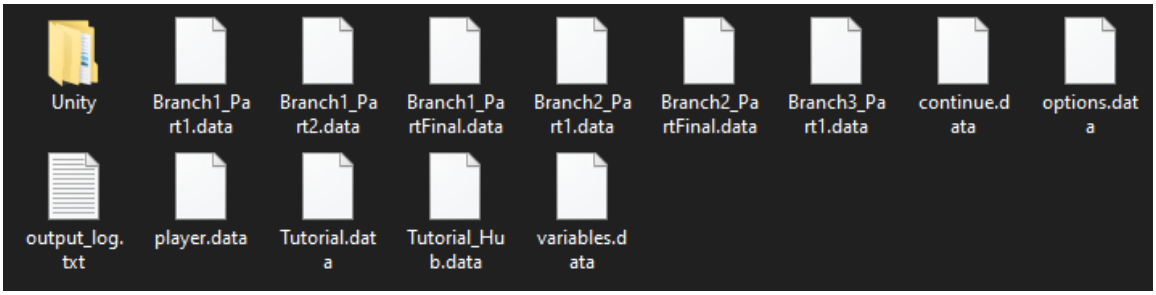
Kuva 12. Lataamiseen käytettävä LoadJson -metodi ja sen rakenne

Kun tieto halutaan myöhemmin käytettävään muotoon, se haetaan tiedostosta ja muutetaan takaisin luokkaan. Kuvassa 12 on käytettävän LoadJson-metodin rakenne. LoadJson-metodin aluksi suoritetaan samanlainen tarkistus, kuin binäärijärjestelmämuodossa. Jos tiedostopolkua tai tiedostoa ei ole, kirjoitetaan Unityn konsoliin viesti. Jos molemmat löydetään, voidaan aloittaa tiedon lataaminen. Tarkistuksen jälkeen luetaan tallennustiedosto ja asetetaan tiedoston teksti uudelle loadedData-muuttujalle. Tiedosto luetaan File.ReadAllText-metodilla, jolle on annettu tiedostopolku, -nimi ja -pääte [21]. Tietojen lukemisen ja asettamisen jälkeen voidaan luoda uusi luokka, johon tiedot muutetaan käytettävään muotoon. Uutta JsonData-luokkaa luodessa, rakentajametodin sijasta käytetään Unityn JsonUtility.FromJson-metodia. FromJson-metodille annetaan, minkä tyyppiseen objektiin luettu kirjainjono tullaan muuttamaan [10]. Kun tieto on muutettu, voidaan sitä käyttää jälleen tarvittavassa kohteessa. FromJson-metodin sijasta voidaan myös käyttää FromJsonOverwrite-metodia, jota ei tässä esimerkissä käytetty. Metodit eroavat toisistaan siten, kun FromJson-metodi luo uuden objektin, FromJsonOverwrite-metodi vain asettaa annetun objektin muuttujiin ladatut arvot [10, 22]. Kuvassa 13 on kirjoitettu tiedostosta ladatut tiedot Unityn konsoliin.

Kuva 13. Unityn konsolin viesti, kun tiedot on ladattu ja kutsuttu JsonData-luokan DebugData-metodia.

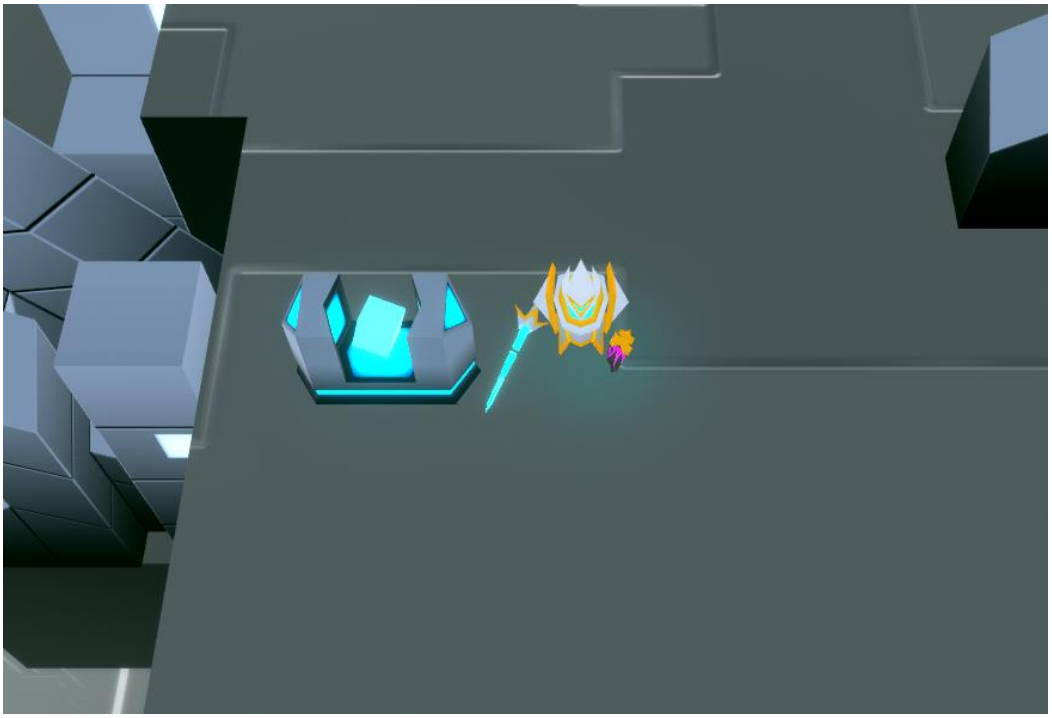
4 Tiedon tallentaminen ja lataaminen CONTINUE-pelissä

CONTINUE-peli tallentaa pelistä monenlaisia eri asioita. Osa tallennettavista asioista ovat, pelaajan tiedot, vihollisten ja tiettyjen objektien tilat, sekä pelin asetukset. Tämä opinnäytetyö ei käsittele pelin asetusten tallentamista. Tilanteita, missä pelaajan ja kentän tiedot tallennetaan ovat, pelaajan aktivoidessa tallennuspiste, pelaajan liikkuessa kenttien välillä ja erityistilanteet, kuten pomotaistelun voitettua, tietyn objektin aktivoinnissa ja kerättyä päivitysesineen. Pelaajan ja jokaisen kentän tiedot tallennetaan omaan tiedostoon, koska projektin alussa todettiin, että tämä olisi helpoin tapa testata ja kehittää järjestelmää eteenpäin. Myös kehitettäessä koettiin, että tämä on helpoin tapa ladata oikea tiedosto, kun kentän tietoja on ladattava. Kun pelaaja jatkaa peliä, käytetään kentän tietojen lataamiseen erillistä tiedostoa, johon on tallennettu, mihin kenttään pelaaja jäi pelistä poistuessa. Kuvassa 14 näkyy tiedostot, mitkä peli on luonut sitä pelatessa.



Kuva 14. CONTINUE:n tallennuskansio, kun peliä on pelattu muutama kenttä

Ennen kuin tietoja tallennetaan, tehdään kenttään saapuessa haku, missä etsitään kaikki tallennettavat objektit kentästä. Ilman tätä menetelmää, järjestelmää tehdessä huomattiin ongelma, että jotkin objektit saattoivat jäädä hakematta tai päivittämättä, jos objektin tila ei ollut aktiivinen. Kun pelaaja aktivoi tallennuspisteen tai on tallennusta vaativassa tilanteessa, aloitetaan tiedon tallentaminen. Kuvassa 15 on esimerkkitilanne, milloin tallennus tehdään.



Kuva 15. Pelaaja seisoo aktivoidun tallennuspisteen vieressä

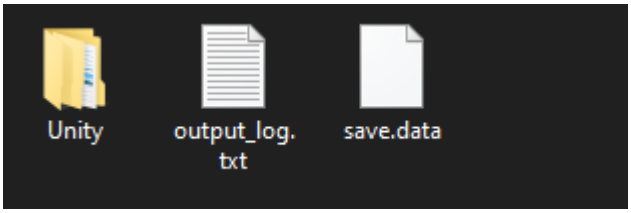
Ensin tallennetaan pelaajan ja tason tiedot. Tiedot haetaan pelaajan ja kentän omiin luokkiin. Kun tiedot on asetettu luokkiin, tehdään tarkistus, onko pelaajan ja kentän tallennustiedosto jo olemassa. Jos tiedosto on olemassa, se voidaan avata ja vanhojen tietojen päälle kirjoittaa. Jos tiedostoa ei ole olemassa, luodaan puuttuva tiedosto. Tiedosto luodaan ja etsitään Unityn luomasta kansiosta käyttämällä Application.persistentDataPath-muuttujaa haun ja luonnin yhteydessä [23]. Kun tiedosto on valmis, luodaan BinaryFormatter-objekti, jota käytetään tiedon muuttamiseen binäärijärjestelmän muotoon [16]. Kun tieto on muutettu, tallennetaan se tiedostoihin ja tiedostot suljetaan. Kun pelaajan ja kentän tiedot on tallennettu, tehdään jatkamiseen tarkoitettu tiedosto. Kentän nimi haetaan ja tallennetaan jatkamistiedostoon samalla tavalla kuin pelaaja ja kenttä.

Peliä jatkaessa tai kentästä toiseen liikkuessa ladataan tallennetut tiedot ja asetetaan ne pelille. Jos peliä jatketaan päävalikosta, ladataan kenttä pelin jatkamiseen luodun tiedoston perusteella. Jatkamistiedosto etsitään, luodaan uusi BinaryFormatter-muuttuja ja muutetaan tieto käytettävään muotoon ja ladataan kenttä saadun arvon perusteella [16]. Jos peliä ei jatketa päävalikosta, vaan pelaaja liikkuu kentästä toiseen, saadaan ladattavan kentän nimi, kun kenttä on latautunut. Ennen kuin kentän ja pelaajan tietoja ladataan, haetaan kentästä tarvittavat objektit ladattujen arvojen asettamiseen. Tämän jälkeen etsitään tiedosto käyttämällä saatua kentän arvoa. Jos tie-

dostoa ei löydy, tason tietoja ei tarvitse ladata. Jos tiedosto on olemassa, luodaan BinaryFormatter-objekti, jota käytetään tiedon muuttamiseen käytettävään muotoon, suljetaan tallennustiedostot ja asetetaan saadut tiedot [16]. Sama tehdään myös pelaajan tiedoille.

5 Tiedon tallentaminen ja lataaminen Quest for Conquest-pelissä

Quest for Conquest-pelissä tallennetaan tiedot yhteen tiedostoon. Tietoja ovat mm. pelaajan resurssit, sekä pelaajan joukkojen ja leirin tila. Pelaajan tietojen lisäksi peli tallentaa kartassa vieraillut kohteet, kartalta löydetyt alueet, pelin aika ja erikoistapahtumiin tarvittavat arvot. Tilanteita, milloin peli tallennetaan, ovat mm., kun pelaaja on suorittanut tapahtuman (kuva 17), pelaaja on voittanut taistelun tai pelaaja saapuu jo vierailtuun tai tyhjään kohteeseen kartalla. Quest for Conquestissa tallennetaan vain yhteen tiedostoon, koska JSON-muoto teki siitä helppoa ja CONTINUE-peliprojektin aikana todettiin, että useampaan tiedostoon tallentaminen on työläämpi vaihtoehto (kuva 16). JSON-muoto valittiin myös, koska projektissa haluttiin oppia siitä samoin, kuin CONTINUE-projektissa binäärijärjestelmämuodosta.



Kuva 16. Quest for Conquest -pelin tallennuskansio

Kun tietoa aletaan tallentaa, haetaan pelin eri managereista tarvittavat tiedot ScriptableObject-tyyppiseen objektiin. Eri managerit sisältävät metodeja, mitkä helpottavat tietojen hakemista objektiin. Kun tarvittavat tiedot on noudettu, muutetaan objekti string-muotoon. Objektin muuttamiseen käytetään Unityn JsonUtility.ToJson-metodia. Kun objekti on string-muodossa, se voidaan kirjoittaa tiedostoon. Tiedoston kirjoittamiseen käytetään File.WriteAllText-metodia, jolle annetaan parametreiksi tiedostopolku, mihin tieto tallennetaan, sekä saatu string-muuttuja, johon luokka on muutettu. Metodia käyttäessä ei myöskään tarvitse huolehtia tiedoston luomisesta, sen avaamisesta tai sulkemisesta, koska metodi hoitaa sen automaattisesti [20].

Kuva 17. Tapahtuma, mikä aktivoi pelin tallennuksen, kun sen loppuun on päästy

Kun tietoja on ladattava peliä jatkaessa, on aluksi tehtävä tarkistus, onko tiedosto olemassa. Jos tiedosto on olemassa, voidaan aloittaa lataus. Tiedosto saadaan helposti käytettävään muotoon käyttämällä JsonUtility.FromJsonOverwrite-metodia. Metodi tarvitsee kaksi parametria, jotta tiedot voidaan muuttaa, tiedostopolku tallennustiedostoon sekä mihin objektiin tiedot halutaan asettaa [22]. Kun tiedot on ladattu ja asetettu, kutsutaan eri managerien metodeja, jotka hakevat ladatut tiedot objektista ja asetetaan ne.

Koska Quest for Conquestin tallennusjärjestelmä tallentaa tiedot JSON-muotoon, tietoja on helppo muokata tekstinmuokkaamistyökaluilla. Tämä ei kuitenkaan ole ongelma, koska peli on vain yksinpelattava ja peli ei sisällä esim. saavutuksia, joita tiedostoa muokkaamalla pelaaja voisi huijata itselleen.

6 CONTINUE ja Quest for Conquestin järjestelmien vertailua

Suurimmat erot CONTINUE:n ja Quest for Conquestin välillä ovat muodot, mihin tieto tallennetaan, sekä CONTINUE:n useampi tallennustiedosto.

CONTINUE-projektin alussa uskottiin, että tietojen salaaminen pelaajalta olisi paras vaihtoehto tallennusjärjestelmää tehdessä. Vaikka pelaaja ei voi pelissä helposti huijata, järjestelmää suunnitellessa ja tehdessä törmättiin useisiin eri ongelmiin ja tämän myötä, järjestelmän toteutus kesti paljon pitempään, mitä aluksi oletettiin. Kaksi suurinta ongelmaa olivat järjestelmään uuden tallennettavan asian lisääminen, sekä järjestelmän testaaminen. Jokainen uusi lisättävä asia toi tallennusjärjestelmään paljon testaamista ja koska testaamista ei voinut tehdä vain lukemalla tietoja kirjoittamisen jälkeen, oli keksittävä uusia tapoja tarkistaa, onko tieto tallentunut oikein. Verrattuna CONTINUE:n järjestelmään, Quest for Conquestin järjestelmä saatiin valmiiksi huomattavasti nopeammin. Vaikka JSON ei salaakaan tietoa, ei sitä vaadita pelissä. Myös valitsemalla JSON binäärijärjestelmämuodon sijasta, opittiin näiden eri muotojen eroja.

CONTINUE:n alussa todettiin parhaaksi tavaksi tallentaa tasojen tieto eri tiedostoihin, koska oikean tiedoston lataaminen olisi helpompaa kenttien välillä liikkuessa. Myös pelin jatkaminen olisi helpompi toteuttaa, jos tiedostoja olisi useampi. Näin järjestelmän testaamisenkin uskottiin olevan helpompaa, kuin tietoja ei tallenneta yhteen tiedostoon. Quest for Conquest-pelin tiedot saatiin helposti haettua yhteen tiedostoon, joten useampaan tiedostoon ei tarvittu.

Järjestelmillä on myös eroja siinä, milloin pelit tallentavat. CONTINUE hakee kaikki tärkeät objektit kentästä, kun pelaaja saapuu kenttään. Tällöin voidaan välttyä ylimääräiseltä haulta, kun tietoa pitäisi tallentaa. Quest for Conquest hakee tiedot pelin eri managerien avulla. Managerit sisältävät metodeja tietojen tallentamiseen ja lataamiseen. CONTINUE tallentaa tietoja, kun pelaaja aktivoi tallennuspisteen tai liikkuu tasojen välillä, kun taas Quest for Conquest tallentaa, kun kartan eri kohteisiin asetettu tapahtuma on suoritettu tai pelaaja on voittanut taistelun.

7 Miten parantaisin CONTINUE:n ja Quest for Conquestin järjestelmiä

Molemmat, CONTINUE ja Quest for Conquest, tarvitsisivat optimointia tallennus- ja latausjärjestelmien puolelta. Koska aikaisempaa kokemusta järjestelmistä ei löytynyt ja nyt kahden projektin jälkeen, järjestelmissä löytyisi paljon optimoitavaa. Myös, milloin järjestelmä tallentaa ja lataa tiedot, olisi hyvä tarkistaa ja verrata, olisiko jokin muu hetki tai tapahtuma parempi tallennukselle. Optimoinnilla saataisiin järjestelmistä kevyempiä ja toimivampia, verrattuna nykyiseen tilaan. Myös turhat tallennukset saataisiin pois CONTINUE:n järjestelmästä optimoinnilla.

CONTINUE-pelissä olisi hyvä muuttaa useampi tiedosto yhdeksi tiedostoksi. Useamman tiedoston sijaan, yksi tiedosto voisi sisältää listan jokaisesta kentästä ja näin pääsisi eroon useasta tiedostosta. Tämä myös avaisi mahdollisuuden tehdä järjestelmästä tukemaan useampaa tallennusta samanaikaisesti. Vaikka Quest for Conquest tallentaakin vain yhteen tiedostoon, peli ei silti tue useampaa tallennusta samanaikaisesti. Jos järjestelmää muuttaisi siten, että se tukisi useampaa tallennusta, tämä varmasti parantaisi peliä.

CONTINUE:n tallennetut asetukset olisi hyvä muuttaa muokattavaan muotoon. Peli tallentaa asetukset binäärijärjestelmämuotoon ja niitä ei sen vuoksi voi muokata pelin ulkopuolella. Hyviä vaihtoehtoja olisivat JSON-muoto tai PlayerPrefs Unityssä. Tästä olisi apua, jos esim. pelaaja ei pääse muuttamaan asetuksia pelin sisällä, jos esimerkiksi pelillä on väärä resoluutio ja vaihtoehtoa ei näe päävalikossa.

8 Yhteenveto

Minkälainen tallennus- ja latausjärjestelmä sopii tietynlaiseen projektiin, riippuu projektin vaatimuksista. Jos projekti on yksinpelattava videopeli, tietoja ei tarvitse välttämättä salata ja ne voi tallentaa pelaajalle paikallisesti. Jos pelissä on minkäänlaista interaktiota pelaajien välillä, tieto on hyvä tallentaa salattuna, pelaajan ulottumattomiin. Myös eri tallennustyypit sopivat paremmin tietyntyyppisiin projekteihin. Binäärijärjestelmän muoto sopii videopeliin, jossa ei haluta pelaajan muokkaavan tallennustiedostoa, kun taas tietokantaan voi tallentaa suuren verkkomoninpelin pelaajahahmojen tiedot.

CONTINUE tallentaa tietoja useaan eri tiedostoon, koska projektia tehdessä todettiin, että tämä olisi paras vaihtoehto toteuttaa tallentaminen. Tiedot tallennetaan binäärijärjestelmän muotoon, koska pelaajan ei haluta pääsevän muokkaamaan tietoa. Tallennettavat objektit haetaan kentän alussa ja ladattaessa annetaan samoille haetuille objekteille ladatut tiedot.

Quest for Conquest-pelin tiedot tallennetaan yhteen tiedostoon ja JSON-muotoon. Tämän projektin alussa haluttiin testata, miten JSON toimii verrattuna binäärijärjestelmän muotoon. Tietojen ollessa JSON-muodossa, pelaajan on mahdollista muokata niitä ja muuttaa videopelin tilannetta. Tiedot haetaan tallennettavaksi eri pelin managerien avulla ja asetetaan latauksen jälkeen samojen managerien avulla.

CONTINUE:n järjestelmä tallentaa useaan eri tiedostoon binäärijärjestelmämuodossa, kun taas Quest for Conquest tallentaa yhteen tiedostoon JSON-muodossa. Myös pelien tallennushetkillä, on eroja. CONTINUE tallentaa yleisesti, kun pelaaja on aktivoinut tallennuspisteen, mutta myös erikoistilanteista on, milloin järjestelmä tallentaa, kun on tarve. Quest for Conquest tallentaa eri tilanteissa, kuten pelaajan suorittaessa tapahtuman tai voittaessa taistelun. Vaikka CONTINUE tallentaakin tiedot binäärijärjestelmän muotoon, järjestelmän suunnittelu ja toteuttaminen veivät huomattavasti enemmän aikaa verrattuna Quest for Conquestin tallennus- ja latausjärjestelmään ja JSON-muotoon.

Molemmat järjestelmät vaatisivat optimointia. CONTINUE-pelin useampi tiedosto olisi hyvä muuttaa yhdeksi tiedostoksi ja mahdollisesti molemmat järjestelmät voisi muuttaa tukemaan useampaa tallennusta samanaikaisesti. Myös CONTINUE:n asetukset olisi hyvä muuttaa muokattavaan muotoon, esimerkiksi JSON tai Unityn PlayerPrefs.

Lähteet

1 Rogers, S. (2010). *Level up!: the guide to great video game design* (p. 492). Chichester: Wiley.

2 Steam Cloud (Steamworks Documentation). (n.d.). Retrieved 11 May 2020, from https://partner.steamgames.com/doc/features/cloud

3 Games Made With Unity. (n.d.). Retrieved 12 May 2020, from https://unity.com/madewith

4 Multiplatform | Unity. (n.d.). Retrieved 14 May 2020, from https://unity.com/features/multiplatform

5 Create and Monetize Games With Unity Gaming Solutions. (n.d.). Retrieved 12 May 2020, from https://unity.com/solutions/game

6 Unity Learn - Supporting Home Learning during COVID-19. (n.d.). Retrieved 14 May 2020, from https://learn.unity.com/

7 Unity - Manual: Unity User Manual (2019.3). (n.d.). Retrieved 14 May 2020, from https://docs.unity3d.com/Manual/index.html

8 Powerful 2D, 3D, VR, & AR software for cross-platform development of games and mobile apps. (n.d.). Retrieved 14 May 2020, from https://store.unity.com/

9 Technologies, U. (2020b). Unity - Scripting API: JsonUtility.ToJson. Retrieved 16 April 2020, from https://docs.unity3d.com/ScriptReference/JsonUtility.ToJson.html

10 Unity - Scripting API: JsonUtility.FromJson. (n.d.). Retrieved 14 May 2020, from https://docs.unity3d.com/ScriptReference/JsonUtility.FromJson.html

11 Binary serialization | Microsoft Docs. (2018). Retrieved 12 May 2020, from https://docs.microsoft.com/en-us/dotnet/standard/serialization/binary-serialization

12 Unity - Manual: ScriptableObject. (2018). Retrieved 12 May 2020, from https://docs.unity3d.com/Manual/class-ScriptableObject.html

13 Unity - Scripting API: PlayerPrefs. (2020). Retrieved 12 May 2020, from https://docs.unity3d.com/ScriptReference/PlayerPrefs.html

14 FileStream Class (System.IO) | Microsoft Docs. (n.d.). Retrieved 12 May 2020, from https://docs.microsoft.com/en-us/dotnet/api/system.io.filestream?view=netframework-4.8

15 FileMode Enum (System.IO) | Microsoft Docs. (n.d.). Retrieved 12 May 2020, from https://docs.microsoft.com/en-us/dotnet/api/system.io.filemode?view=netcore-3.1

16 BinaryFormatter Class (System.Runtime.Serialization.Formatters.Binary) | Microsoft Docs. (2020). Retrieved 16 April 2020, from https://docs.microsoft.com/en-us/dotnet/api/system.runtime.serialization.formatters.binary.binaryformatter?view=netframework-4.8

17 BinaryFormatter.Serialize Method (System.Runtime.Serialization.Formatters.Binary) | Microsoft Docs. (2020). Retrieved 16 April 2020, from https://docs.microsoft.com/en-us/dotnet/api/system.runtime.serialization.formatters.binary.binaryformatter.serialize?view=netframework-4.8

18 BinaryFormatter.Deserialize Method (System.Runtime.Serialization.Formatters.Binary) | Microsoft Docs. (2020). Retrieved 16 April 2020, from https://docs.microsoft.com/en-us/dotnet/api/system.runtime.serialization.formatters.binary.binaryformatter.deserialize?view=netframework-4.8

19 Unity - Scripting API: SerializeField. (n.d.). Retrieved 14 May 2020, from https://docs.unity3d.com/ScriptReference/SerializeField.html

20 File.WriteAllText Method (System.IO) | Microsoft Docs. (2020). Retrieved 16 April 2020, from https://docs.microsoft.com/en-us/dotnet/api/system.io.file.writealltext?view=netframework-4.8

21 File.ReadAllText Method (System.IO) | Microsoft Docs. (2020). Retrieved 16 April 2020, from https://docs.microsoft.com/en-us/dotnet/api/system.io.file.readalltext?view=netframework-4.8

22 Unity - Scripting API: JsonUtility.FromJsonOverwrite. (n.d.). Retrieved 14 May 2020, from https://docs.unity3d.com/ScriptReference/JsonUtility.FromJsonOverwrite.html

23 Technologies, U. (2020a). Unity - Scripting API: Application.persistentDataPath. Retrieved 6 April 2020, from https://docs.unity3d.com/ScriptReference/Application-persistentDataPath.html