Tampereen yliopisto

Hermanni Rytkölä

PELITILAN TALLENNUS JA LATAUS UNITY-PELIMOOTTORILLA

Kandidaatintyö
Informaatioteknologian ja viestinnän tiedekunta
Tarkastajat: Petri Kannisto
Huhtikuu 2021

TIIVISTELMÄ

Hermanni Rytkölä: Pelitilan tallennus ja lataus Unity-pelimoottorilla
Kandidaatintyö
Tampereen yliopisto
Tutkinto-ohjelma
Huhtikuu 2021

Videopelin tilan tallentaminen on pelaajille tärkeä ominaisuus. Pelaaja pystyy keskeyttämään pelin ja jatkamaan sitä myöhemmin ilman, että peli on aloitettava kokonaan alusta. Pelin tilan tallentaminen on monimutkainen ja virhealtis prosessi. Ohjelmoijan on minimoitava tallennettavan datan määrä, jotta tallennustiedoston koko sekä tallennukseen ja lataukseen kuluva aika saadaan mahdollisimman alhaiseksi.

Tässä kandidaatintyössä tutkittiin tallennusjärjestelmien integrointia pelimaailmaan sekä tallennusjärjestelmän toteuttamismahdollisuuksia Unity-pelimoottorilla. Unity-pelimoottorilla pelitilan tallennus voidaan toteuttaa serialisaation avulla. Pelimaailmasta serialisoidaan kaikki pelitilan kannalta välttämättömät peliobjektit. Pelitila voidaan myöhemmin palauttaa deserialisoimalla tallennettu data uusiin peliobjekteihin.

Työssä tallennettiin yksinkertainen Unity-pelimoottorin peliobjekti levylle binääri-, JSON(JavaScript Object Notation)- sekä XML(Extensible Markup Language)- muotoisena tiedostona. Eri tiedostomuotojen tallennus- ja latausnopeutta sekä tallennustiedostojen kokoa vertailtiin. Työssä tehdyissä mittauksissa nopeimmat tallennus- ja latausajat tallennustiedostolle saatiin serialisoimalla tiedosto binäärimuotoon. Tallennustiedoston koko saatiin minimoitua serialisoimalla tiedosto JSON-muotoon Unityn valmiin `JsonUtility`-luokan avulla. Kun tallennettavaa dataa on vähän, eri serialisaatiomenetelmien väliset erot ovat kuitenkin pelaajan näkökulmasta hyvin pieniä. Tämän myötä serialisaatiomenetelmän ja tallennustiedostomuodon valinnalla ei ole pelikokemuksen kannalta peliprojektissa suurta merkitystä, ellei tallennettavaa dataa ole usean megatavun verran.

Pelikonfiguraatio eli peliasetukset voidaan tallentaa käyttämällä Unityn valmista PlayerPrefs-luokkaa. PlayerPrefs ei kuitenkaan sovellu varsinaisen pelitilan tallentamiseen, vaan siihen suositellaan sen sijaan serialisaation käyttöä.

Unitylle on saatavilla useita lisäosia, jotka helpottavat tallennusjärjestelmän integrointia peliin. Lisäosien avulla ohjelmoija voi tallentaa kokonaisia peliobjekteja yhdellä koodirivillä. Ohjelmoijan ei siis tarvitse perehtyä tarkemmin tallennusprosessissa tapahtuvaan serialisaatioon ja deserialisaatioon, mutta hänen on silti mietittävä, mitä kaikkea pelimaailmasta on tallennettava.

Avainsanat: tallennus- ja latausjärjestelmät, serialisaatio, videopelit

SISÄLLYSLUETTELO

LYHENTEET JA MERKINNÄT

.NET Framework	Microsoftin kehittämä ohjelmistokehys
C#	C sharp, .NET-alustaan pohjautuva ohjelmointikieli
JSON	JavaScript Object Notation, rakenteellisen datan tallennukseen soveltuva tiedostomuoto
XML	Extensible Markup Language, rakenteellisen datan tallennukseen soveltuva tiedostomuoto

1. JOHDANTO

Pelaajan pelikokemuksen kannalta on tärkeää, että videopeleissä on jonkinlainen tallennustoiminto. Modernit pelimoottorit pystyvät simuloimaan lukemattomia peliobjekteja samanaikaisesti. Pelaajat voivat muokata pelimaailmaa mielensä mukaan ja aiheuttaa pelimaailmaan ennalta-arvaamattomia muutoksia. Pelit ovat lisäksi pituudeltaan niin laajoja, ettei niitä voi pelata yhdessä sessiossa läpi. On siis tärkeää, että pelin voi tarvittaessa keskeyttää ja että samasta pelitilanteesta voisi myöhemmin myös jatkaa. Tämä voidaan mahdollistaa lisäämällä peliin tallennusominaisuus, jonka avulla pelaaja voi tallentaa hetkellisen pelitilan.

Pelitilan tallentamista varten tarvitaan järjestelmä, jonka avulla pelin hetkellisestä tilasta voidaan luoda kopio. Tallennetun pelitilan voisi sitten myöhemmin palauttaa kyseisen kopion avulla. Tietokoneohjelman tilan täydellinen tallentaminen ei kuitenkaan ole helppo asia. Kaiken ladatun datan tallentaminen ei ole käytännöllistä, sillä se johtaisi kohtuuttoman suurikokoisiin tallennustiedostoihin sekä pidentäisi tallennukseen ja lataukseen kuluvaa aikaa. Pelin kehittäjän on määritettävä, mikä data on pelitilan säilyttämisen kannalta tärkeää ja mikä ei.

Työn tavoitteena on tutkia, miten videopelien dataa tallennetaan ja ladataan ja miten eri tallennus- ja latausratkaisut eroavat toisistaan. Työ tarkastelee erityisesti Unity-pelimoottorille saatavilla olevia tallennusjärjestelmäratkaisuja. Vaikka lähes kaikki uudet pelit sisältävät tallennus- ja latausjärjestelmän, on aiheesta varsin vähän tietoa saatavilla. Tämä työ pyrkii kokoamaan aiheesta saatavilla olevat tiedot yhteen.

Työn toisessa luvussa tutkitaan, millä tavoin tallennusjärjestelmä on mahdollista integroida peliin. Lisäksi toisessa luvussa pohditaan, minkälaista dataa pelistä on tallennettava ja minkälaiseen muotoon dataa kannattaa tallentaa. Kolmannessa luvussa tarkastellaan tallennusjärjestelmän toteutusta erityisesti Unity-pelimoottorilla kehitettävien pelien kannalta. Neljännessä luvussa luodaan Unity-pelimoottorissa yksinkertainen peliobjekti ja suoritetaan objektin tallennus- ja latausoperaatio eri menetelmillä. Viides luku sisältää yhteenvedon työn havainnoista ja johtopäätöksistä.

2. TALLENNETTAVA DATA JA TALLENNUSTAVAT

2.1 Tallennettava data

Tallennettava data voidaan luokitella eri ryhmiin, jotka tallennetaan eri tavalla ja joiden tallennukselle asetetaan erilainen prioriteetti. Yksi tapa on jakaa data kahteen ryhmään: käyttäjäkonfiguraatioon sekä pelidataan [1]. Käyttäjäkonfiguraatio sisältää kaikki pelin tilasta riippumattomat tiedot. Tämänkaltaisia tietoja ovat esimerkiksi pelin äänenvoimakkuusasetukset sekä tieto siitä, onko peli-ikkuna koko näytön tilassa.

Pelidata sisältää käytännössä kaiken, mitä pelaaja kokee ja näkee [2]. Kaiken tämän datan tallentaminen veisi kuitenkin paljon levytilaa, joten vain pelin toiminnan kannalta tärkeät asiat kannattaa tallentaa [3]. Tallennettavasta datasta voidaan jättää pois kaikki sellainen tieto, jota peli pystyy itse päättelemään ladatusta pelitilasta. [4] Esimerkiksi pelihahmojen hetkellisen rotaation tallentaminen ei aina ole välttämätöntä. Pelihahmoille voidaan tallentamisen sijaan arpoa satunnainen rotaatio kun pelitila palautetaan [3].

2.2 Tallennustavat

Pelaajalle on tärkeää, että pelin voi keskeyttää milloin tahansa ja että peliä voisi myöhemmin myös jatkaa samasta kohdasta [2] [5] [6] [7]. Täydellisen tallennusjärjestelmän toteuttaminen on kuitenkin ohjelmoijalle haasteellinen tehtävä [2] [8]. Tallennusjärjestelmää suunniteltaessa on tärkeää rajata tarkkaan, missä tilanteissa pelaaja voi tallentaa pelin [2]. Tässä osiossa esitellään yleisiä ratkaisuja siihen, miten tallennustoiminto voidaan integroida peliin.

Pelin kehityksen alkuvaiheissa on hyvä kartoittaa etukäteen tilanteet, joissa pelin tilan tallentaminen voisi olla haastavaa. Tallentaminen kannattaa sallia vain silloin, kun pelin tila on vakaa. Esimerkiksi seikkailupelissä tallentaminen saattaa olla mahdotonta sellaisena hetkenä, jonka aikana pelikartassa tapahtuu suuria muutoksia. Vuoropohjaisessa strategiapelissä samankaltainen tilanne saattaisi tulla eteen, kun tekoälypelaaja toteuttaa omaa vuoroaan tai kun pelimaailmassa on käynnissä taistelu, joka vaikuttaa useiden peliobjektien tilaan. [3]

80- ja 90-luvun peleissä pelaajan edistyksestä pidettiin kirjaa usein salasanan avulla [9].

Peli voidaan jakaa tasoihin, joille annetaan tietty salasana. Kun pelaaja saavuttaa pelissä seuraavan tason, peli paljastaa pelaajalle salasanan, jonka avulla pelaaja voi myöhemmin palata samalle tasolle pelin aloitusvalikosta. Salasanajärjestelmä ei voi kuitenkaan tallentaa nykyistä pelitilaa, vaan ainoastaan pelaajan edistyksen. Peliä ei siis voi tallentaa kesken tason, eivätkä yhdellä tasolla tapahtuvat muutokset säily seuraavalle tasolle. [8]

Monet pelit tallentavat pelitilan automaattisesti tietyin väliajoin [10]. Pelaaja voi turvautua automaattitallennuksen luomaan tallennustiedostoon, jos hän on unohtanut tallentaa pelin tai jos peli on yllättäen kaatunut. Automaattitallennus voidaan suorittaa esimerkiksi aina tietyin väliajoin tai kun pelaaja on ylittänyt pelimaailmassa jonkinlaisen rajan. [11] Automaattisesti luotu tallennustiedosto ei yleensä korvaa pelaajan itse tekemiä tallennustiedostoja.

Useissa peleissä on käytössä tallennuspistejärjestelmä [2] [12]. Pelaaja voi tallentaa pelin vain tietyissä, ennalta määritetyissä paikoissa. [10] Ohjelmoija voi sijoittaa tallennuspisteet pelimaailmaan siten, että pakollisesti tallennettavaa dataa on tallennushetkellä mahdollisimman vähän. Hyvä esimerkki tallennuspisteiden käytöstä on *Resident Evil* -pelisarjassa, jossa pelaaja voi tallentaa pelin kirjoituskoneiden luona. [11] Kirjoituskoneet on aina sijoitettu pelimaailmaan siten, ettei niiden lähistöllä ole vihollisia tai muita pelitilan kannalta oleellisia peliobjekteja, joiden tilasta olisi pidettävä kirjaa [10].

Monissa peleissä pelin voi tallentaa missä tahansa ja milloin tahansa yhdellä napinpainalluksella [13]. Esimerkiksi *The Elder Scrolls* -pelisarja tarjoaa pelaajalle quicksave- ja quickload- toiminnot, joiden avulla pelaaja voi pikaisesti tallentaa tai ladata pelin painamalla F5- ja F9- näppäimiä. Uuden quicksaven luominen korvaa aiemmin luodun quicksave-tallennustiedoston. Pelaajia kehotetaan luomaan varmuuden vuoksi quicksave-tallennusten lisäksi myös tavallisia tallennustiedostoja tasaisin väliajoin. [14] Quicksave-ominaisuuden toteutus on ohjelmoijalle haasteellinen tehtävä, sillä pelin tallentamisen on oltava lähes aina mahdollista riippumatta siitä, mitä pelimaailmassa tapahtuu.

Tallennustavoista on koottu karkea yhteenveto taulukkoon 2.1. Taulukko vertailee tallennustapoja kahdella kriteerillä: missä ja milloin pelin voi tallentaa, ja miten haastavaa tallennustavan toteutus on ohjelmoijan näkökulmasta.

Taulukko 2.1. *Yleisimmät tallennustavat koottuna*

Tallennustapa	**Tallennusmahdollisuus**	**Haastavuus**
Tallennuspisteet	Tietyissä paikoissa	Keskisuuri
Automaattitallennus	Tietyissä paikoissa/missä vain	Keskisuuri/vaikea
Quicksave	Missä vain	Vaikea
Tasoihin jako/Salasana	Tasojen välissä	Helppo

2.3 Tallennustiedostot

Monimutkaisissa peliprojekteissa on usein tarpeellista turvautua tiedostomuotoihin, joissa tallennettujen peliobjektien tai tietorakenteiden muoto säilyy. Dataa voidaan tallentaa levylle binäärimuodossa tai tekstimuodossa. Binäärimuodossa data tallennetaan sarjaksi bittejä, joiden arvo on aina joko yksi tai nolla. Ihmiselle binääridata ei ole lukukelpoista, mutta tietokoneet pystyvät tulkitsemaan sitä. [15]

Tekstimuotoiset tiedostotyypit, kuten XML (Extensible Markup Language) ja JSON (JavaScript Object Notation), ovat sen sijaan ihmiselle lukukelpoisia. Ohjelmassa 2.1 on esitetty kuvitteellinen tietorakenne JSON-muodossa. Ohjelmassa 2.2 on esitetty sama tietorakenne XML-muodossa.

Pelidatan tallentaminen binääri-, XML- ja JSON-muotoihin on mahdollista datan *serialisaation* avulla. [16] Serialisaatio on prosessi, jossa olio tai tietorakenne tallennetaan teksti- tai binääridatamuotoon. Levylle serialisoitu olio voidaan *deserialisoida*, joka on serialisoinnille päinvastainen prosessi. Olion tai tietorakenteen tila voidaan palauttaa tallennetun kaltaiseksi deserialisoimalla tallennettu tiedosto. [17] [18]

```
1  {
2    "Enemy": {
3      "Type": "Frog",
4      "Level": 2,
5      "Health": 25,
6      "Coordinates": {
7        "X": 15,
8        "Y": -7,
9        "Z": 0
10     },
11     "StatusEffects": {
12       "Burning": 1,
13       "Poisoned": 0,
14       "Shocked": 0
15     }
16   }
17 }
```

Ohjelma 2.1. *Esimerkki JSON-muotoon tallennetusta peliobjektista*

```
1     <Enemy>
2    <Type>Frog</Type>
3    <Level>2</Level>
4    <Health>25</Health>
5    <Coordinates>
6           <X>15</X>
7           <Y>–7</Y>
8           <Z>0</Z>
9         </Coordinates>
10      <StatusEffects>
11          <Burning>1</Burning>
12          <Poisoned>0</Poisoned>
13          <Shocked>0</Shocked>
14      </StatusEffects>
15 </Enemy>
```

Ohjelma 2.2. *Esimerkki XML-muotoon tallennetusta peliobjektista*

3. TALLENNUS UNITY-PELIMOOTTORILLA

3.1 Unity

Videopelien kehityksessä käytetään yleensä jonkinlaista pelimoottoria. Pelimoottori on eräänlainen valmis pohja, jonka päälle pelinkehittäjät voivat toteuttaa oman pelinsä. Pelimoottorit tarjoavat yleensä vähintään jonkinlaisen menetelmän peliobjektien piirtämiseen sekä järjestelmän, jonka avulla peliobjektien liikettä voi simuloida pelimaailmassa. [19] Tallennusjärjestelmän toteutus tapahtuu eri pelimoottoreilla eri tavoin. Jotkin pelimoottorit sisältävät valmiin rajapinnan pelidatan tallennusta varten. Tässä työssä tarkastellaan, miten tallennusjärjestelmä voidaan toteuttaa Unity-pelimoottoriin pohjautuvissa peleissä.

Unity on yleisesti käytetty pelimoottori, jonka avulla voidaan kehittää sekä kaksi- että kolmiulotteisia videopelejä useille eri alustoille. [20] Valtaosa Unityllä toteutetuista projekteista ohjelmoidaan C#-kielellä. C# on Microsoftin .NET -ohjelmointialustaan perustuva olio-ohjelmointiin suunnattu ohjelmointikieli. [21] Unity ei sisällä valmista järjestelmää pelidatan tallennusta varten. Pelidatan tallentaminen on kuitenkin mahdollista C#:n serialisaatio-ominaisuuksien avulla. Tässä työssä tarkastellaan datan serialisointia binääri-, XML- sekä JSON-muotoon.

3.2 Serialisaatio

3.2.1 Binääriserialisaatio

C#-pohjaisissa Unity-projekteissa binääriserialisaatioon voidaan käyttää .NET Frameworkiin kuuluvaa `BinaryFormatter`-luokkaa, jonka avulla olio tai tietorakenne voidaan muuttaa binäärimuotoon [18]. Binäärimuotoinen data voidaan sitten kirjoittaa levylle tallennustiedostoon.

Kun aiempi pelitila halutaan palauttaa, prosessi suoritetaan käänteisessä järjestyksessä. `BinaryFormatter` deserialisoi tallennetusta tiedostosta tietorakenteen, joka on identtinen alun perin tallennetun tietorakenteen kanssa. Lopuksi pelimaailmaan luodaan uudet peliobjektit ja niille asetetaan tallennustiedostosta ladatut arvot. [22]

3.2.2 JSON-serialisaatio

Unity sisältää valmiin luokan `JsonUtility`, jonka avulla pelimoottorin olioita voidaan muuttaa JSON-muotoiseksi dataksi ja JSON-dataa olioiksi. Oliosta tai tietorakenteesta tallennettavat ominaisuudet on asetettava julkisiksi `Serializable`-attribuutillla, jotta ne voidaan serialisoida. [23] Myöhemmin esitettävässä ohjelmassa 4.2 on esimerkki attribuutin käytöstä.

Olio tai tietorakenne voidaan muuttaa JSON-muotoiseksi merkkijonoksi `JsonUtility`n metodilla `ToJson`. Tämä merkkijono voidaan sitten tallentaa levylle. JSON-muotoinen tekstitiedosto voidaan vastavuoroisesti deserialisoida `JsonUtility`n metodilla `FromJson`. `FromJson` luo uuden instanssin tallennetusta oliosta ja asettaa olion kentille JSON-tiedostoon tallennetut arvot. [23]

3.2.3 XML-serialisaatio

XML-serialisaatio on mahdollista Unity-projekteissa .NET Frameworkin `XmlSerializer`-luokalla [24]. `XmlSerializer`in toimintaa voidaan ohjata lisäämällä serialisoitavan koodin yhteyteen erilaisia attribuutteja, jotka vaikuttavat XmlSerializerin käyttäytymiseen. [25] Muutoin XML-serialisaatioprosessi ei juuri poikkea JSON-serialisaatiosta.

XML-deserialisaatio on samankaltainen prosessi kuin binääritiedostojen ja JSON-tiedostojen deserialisaatio. `XmlSerializer`ille annetaan XML-muotoinen tiedosto, joka deserialisoidaan. `XmlSerializer` luo uuden olion ja asettaa sen attribuuteille XML-tiedostoon tallennetut arvot. [25]

3.3 PlayerPrefs

Unity tarjoaa kehittäjille sisäänrakennetun `PlayerPrefs`-luokan, johon on mahdollista tallentaa liukulukuja, kokonaislukuja sekä merkkijonoja. Tallennettavalle datalle annetaan tekstiavain, jonka avulla kyseistä dataa voidaan hakea `PlayerPrefs`istä. [26]

`PlayerPrefs`in pääasiallinen tarkoitus on toimia tallennuspaikkana käyttäjäkonfiguraatiolle. Pelidatan tallennukseen `PlayerPrefs` ei ole ideaalinen [22], mutta sitä voidaan käyttää yksinkertaisissa peleissä esimerkiksi pelaajan edistyksen seuraamiseen.

4. ESIMERKKEJÄ

4.1 Esimerkki peliobjektista

Tässä luvussa luodaan Unity-pelimoottorilla uusi peliobjekti, jolle suoritetaan sekä tallennus- että latausoperaatio eri serialisaatiomenetelmillä. Määritellään aluksi sellainen peliobjekti, joka sisältää sekä teksti-, kokonaisluku- että totuusarvomuotoista dataa.

Unity-pelimoottorissa kaikki peliobjektit kuuluvat luokkaan `GameObject`. `GameObject`in käyttäytymistä voi muuttaa liittämällä siihen C#-kielellä kirjoitettuja ohjelmia. [27] Luodaan uusi `GameObject` ja liitetään siihen yksinkertainen ohjelma `Hero`, joka kuvaa sankaria kuvitteellisessa seikkailupelissä. Sankarilla on merkkijonomuotoinen nimi `name`, kokonaislukuina esitettävä määrä elämäpisteitä `health` sekä reppu `inventory`, jonka sisällä on esineitä. Esineillä on tunnusluvut, joista jokainen vastaa tiettyä esinettä pelimaailmassa. Esimerkiksi luku 1 voisi vastata miekkaa ja luku 3 lapiota. Nämä tunnusluvut on tallennettu kokonaislukuina listaan.

Peli tarvitsee myös syystä tai toisesta tiedon siitä, onko sankarin lähellä vihollisia. Tämä tieto tallennetaan totuusarvoon `enemiesNearby`. Sankarilla on myös tietty sijainti pelimaailmassa. Tämä paikkatieto on tallennettu Unityn `Transform`-luokkaan `Vector3`-muotoisena vektorina, jonka komponenttien arvot ovat liukulukuja [28]. Jokainen Unityn `GameObject`-olio sisältää automaattisesti `Transform`-olion, joten sijaintia ei tarvitse tallentaa `Hero`-luokkaan. [29]. Hero-luokan rakenne on esitetty ohjelmassa 4.1.

```
1 public class Hero : MonoBehaviour
2 {
3     private string name;
4     private int health;
5     private List<int> inventory = new List<int>();
6     private bool enemiesNearby;
7 }
```

Ohjelma 4.1. *Hero-luokan rakenne*

`Hero`-luokasta tallennetaan vain pelitilan kannalta kriittiset tiedot. Pelaajan nimi, elämäpisteet ja repun sisältö on pakko tallentaa, jotta pelitila voidaan myöhemmin palauttaa. Totuusarvo `enemiesNearby` voidaan määrittää automaattisesti kun peli on ladattu, joten sen

tallentaminen ei ole välttämätöntä. Sankarin sijainti on myös tallennettava. Luodaan näiden tietojen avulla Hero-luokkaan serialisoitava tietue `HeroStatus`, joka sisältää kaiken tallennettavan datan. Kun peli halutaan tallentaa, `Hero`-luokkan `GetCurrentState()`-metodia kutsuaan, mikä luo uuden `HeroStatus`-tietueen. `Hero` pystyy myös lataamaan aiemman pelitilan metodin `LoadHeroState` avulla. `HeroStatus`-tietueen rakenne on esitetty ohjelmassa 4.2. Ensimmäisellä rivillä oleva attribuutti `[System.Serializable]` merkitsee tietueen serialisoitavaksi [30].

```
1  [System.Serializable]
2  public struct HeroStatus
3  {
4      public string name;
5      public int health;
6      public List<int> inventory;
7      public float x;
8      public float y;
9      public float z;
10 }
11
12 public HeroStatus GetCurrentState()
13 {
14     return new HeroStatus()
15     {
16         name = this.name,
17         health = this.health,
18         inventory = this.inventory,
19         x = transform.position.x,
20         y = transform.position.y,
21         z = transform.position.z
22     };
23 }
24
25 public void LoadHeroState(HeroStatus loadedState)
26 {
27     name = loadedState.name;
28     health = loadedState.health;
29     inventory = loadedState.inventory;
30     transform.position = new Vector3(loadedState.x, loadedState.y
           , loadedState.z);
```

```
31 }
```

Ohjelma 4.2. *HeroStatus-tietueen rakenne ja käyttö*

4.2 Binääriserialisaatio

Tässä osiossa tallennetaan ja ladataan Hero-olio binääriserialisaation avulla. Tätä varten on toteutettava uusi luokka `Save`, joka huolehtii peliobjektien tietojen tallennuksesta ja latauksesta. Luokalla `Save` on sankariolioon `Hero` viite `HeroInstance`, jonka avulla luokka pystyy kutsumaan `Heron` metodeja `GetCurrentState()` sekä `LoadHeroState()`.

Sankarin tila tallennetaan binääritiedostoon `BinaryFormatterin` avulla. Tallennusolio kutsuu sankarin metodia `GetCurrentState()` saadakseen sankarin nykyisen tilan, joka sijaitsee tietueessa `HeroStatus`. Tämän jälkeen `HeroStatus` serialisoidaan binäärimuotoiseksi tiedostoksi. Hero-olion tallennus binääritiedostoon on esitetty ohjelmassa 4.3.

```
1 void SaveBinary()
2 {
3     Hero.HeroStatus status = heroInstance.GetCurrentState();
4     BinaryFormatter binaryFormatter = new BinaryFormatter();
5     FileStream saveFile = File.Create("SaveFiles/save.binary");
6     binaryFormatter.Serialize(saveFile, status);
7     saveFile.Close();
8 }
```

Ohjelma 4.3. *HeroStatuksen tallennus binääritiedostoon*

Datan lataus binääritiedostosta on käytännössä sama prosessi kuin tallentaessa, mutta vastakkaisessa järjestyksessä. Levylle tallennettu binääritiedosto deserialisoidaan, jolloin saadaan `HeroStatus`-tietue. Tämä tietue lähetetään `Hero`-oliolle, joka alustaa lopuksi itsensä ladatun tietueen avulla. Lopputuloksena `Hero` on samassa tilassa kuin se oli alkuperäisen tallennustiedoston luomishetkellä. Binääritiedoston lataus on esitetty ohjelmassa 4.4.

```
1 void LoadBinary()
2 {
3     BinaryFormatter formatter = new BinaryFormatter();
4     FileStream saveFile = File.Open("SaveFiles/save.binary",
          FileMode.Open);
5     Hero.HeroStatus status = (Hero.HeroStatus)formatter.
          Deserialize(saveFile);
6     heroInstance.LoadHeroState(status);
```

```
7       saveFile.Close();
8   }
```

Ohjelma 4.4. *HeroStatuksen lataus binääritiedostosta*

4.3 JSON-serialisaatio

JSON-muotoisen tallennustiedoston luominen on samankaltainen prosessi kuin binääritiedoston luominen. Sankariolion serialisointiin käytetään `BinaryFormatter`in sijaan Unityn toteuttamaa `JsonUtility`ä. Serialisoitu teksti on lisäksi muutettava biteiksi levylle kirjoittamista varten. JSON-tiedostoon tallennus on esitetty ohjelmassa 4.5.

```
1   void SaveJSON()
2   {
3       Hero.HeroStatus status = heroInstance.GetCurrentState();
4       FileStream saveFile = File.Create("SaveFiles/save.json");
5       string jsonData = JsonUtility.ToJson(status);
6       byte[] bytes = Encoding.UTF8.GetBytes(jsonData);
7       saveFile.Write(bytes, 0, bytes.Length);
8       saveFile.Close();
9   }
```

Ohjelma 4.5. *HeroStatuksen tallennus JSON-tiedostoon*

Funktio `LoadJSON()` luo tallennustiedoston avulla uuden `HeroStatus`-tietueen ja lähettää sen sankarioliolle. Latausprosessi ei eroa binääriserialisaation latausprosessista. JSON-tiedoston latausprosessi on esitetty ohjelmassa 4.6.

```
1   void LoadJSON()
2   {
3       string loadedJson = File.ReadAllText("SaveFiles/save.json");
4       Hero.HeroStatus loadedStatus = JsonUtility.FromJson<Hero.
            HeroStatus>(loadedJson);
5       heroInstance.LoadHeroState(loadedStatus);
6   }
```

Ohjelma 4.6. *HeroStatuksen lataus JSON-tiedostosta*

4.4 XML-serialisaatio

XML-serialisaatio on ohjelmoijan kannalta hyvin samankaltainen prosessi kuin binääriserialisaatio. Erona on, että serialisointiin ja deserialisointiin käytetään `BinaryFormatter`in sijaan `XMLSerializer`ia. Tallennus ja lataus on esitetty ohjelmissa 4.7 ja 4.8.

```
1 void SaveXML()
2 {
3     Hero.HeroStatus status = heroInstance.GetCurrentState();
4     XmlSerializer xmlSerializer = new XmlSerializer(status.
          GetType());
5     FileStream saveFile = File.Create("SaveFiles/save.xml");
6     xmlSerializer.Serialize(saveFile, status);
7     saveFile.Close();
8 }
```

Ohjelma 4.7. *HeroStatuksen tallennus XML-tiedostoon*

```
1 void LoadXML()
2 {
3     XmlSerializer xmlSerializer = new XmlSerializer(typeof( Hero.
          HeroStatus));
4     FileStream saveFile = File.Open("SaveFiles/save.xml",
          FileMode.Open);
5     Hero.HeroStatus status = (Hero.HeroStatus)xmlSerializer.
          Deserialize(saveFile);
6     heroInstance.LoadHeroState(status);
7     saveFile.Close();
8 }
```

Ohjelma 4.8. *HeroStatuksen lataus XML-tiedostosta*

4.5 Tallennustiedostojen vertailu

Tässä osiossa tarkastellaan, miten eri serialisaatiomenetelmillä luodut tallennustiedostot käytännössä eroavat toisistaan. Asiaa pohditaan tutkimalla kolmea asiaa: `Hero`-olion tallennukseen kuluvaa aikaa, `Hero`-olion lataukseen kuluvaa aikaa sekä lopullisen peliobjektin kokoa. Näitä ominaisuuksia mitattiin suorittamalla kaikkia tallennus- ja latausfunktioita yhteensä tuhat kertaa. Koska `Hero`-olio sisältää suhteellisen vähän dataa, tehtiin mittaukset kahdella eri tavalla. Toisessa `Hero`-olion reppuun mahtui viisi esinettä, toisessa 10 000. Käytännössä siis `Heron` `inventory`-listaan tallennettiin joko viisi tai 10 000 kokonaislukua.

Kaikki mitatut tulokset kirjattiin ylös ja niistä laskettiin keskiarvo. Funktioiden prosessointiajan mittaukseen käytettiin Unityn `Time`-luokan ominaisuutta `realtimeSinceStartup`. Lasketut keskiarvot on esitetty taulukoissa 4.1, 4.2 ja 4.3.

Taulukko 4.1. *Tallennustiedoston luomiseen kulunut aika, 1 000 toiston keskiarvo*

Repun koko	Binääridata	JSON	XML
5	3,07ms	3,03ms	3,44ms
10 000	4,10ms	4,58ms	11,01ms

Tallennustiedoston luominen on suunnilleen yhtä nopea prosessi kaikilla kolmesta serialisaatiomuodosta tallennettavan objektin ollessa pieni. Suurella repun koolla binääritiedoston luominen on hieman nopeampaa kuin JSON-tiedoston kirjoittaminen. XML-tiedoston tallennukseen taas kului runsaasti muita tiedostomuotoja enemmän aikaa.

Taulukko 4.2. *Tallennustiedoston lataukseen kulunut aika, 1 000 toiston keskiarvo*

Repun koko	Binääridata	JSON	XML
5	0,192ms	0,160ms	0,266ms
10 000	0,279ms	0,860ms	6,394ms

Repun koon ollessa pieni, JSON-muotoisen tiedoston latausprosessiin kului vähiten aikaa. Suuremmalla repun koolla binääritiedoston lataus sujui kuitenkin moninkertaisesti muita vaihtoehtoja nopeammin. XML-tiedoston lataukseen kului huomattava määrä aikaa, kun repun koko oli suuri.

Taulukko 4.3. *Tallennustiedoston vaatima levytila, 1 000 toiston keskiarvo*

Repun koko	Binääridata	JSON	XML
5	538 tavua	122 tavua	378 tavua
10 000	64,4 kt	19,6 kt	176 kt

`Hero`-olio pakkautuu JSON-tiedostomuodossa huomattavasti binääri- ja XML-tiedostoja paremmin. Pienellä repun koolla sekä XML- että JSON-tiedostot ovat binääritiedostoa pienempiä. Repun koon kasvaessa XML-tiedoston koko kasvaa muita tiedostotyyppejä nopeammin.

Mittaustulosten perusteella `Hero`-olion tallentamiseen olisi järkevää käyttää joko binääriserialisaatiota tai JSON-serialisaatiota. Unity-pelimoottori pystyy tulosten perusteella tallentamaan ja lataamaan binääridataa muita tiedostomuotoja nopeammin, varsinkin kun dataa on paljon. Toisaalta JSON-data pakkautui testeissä binääridataa paremmin, mikä säästää pelaajien levytilaa. XML-serialisaatio suoriutui mittauksista selvästi muita serialisaatiomenetelmiä heikommin. XML-tiedoston tallennukseen ja lataukseen kuluva aika on kuitenkin isollakin repun koolla vain muutama millisekunti, mikä ei pelaajan kannalta ole kohtuuttoman pitkä aika. XML-serialisaation käyttöä ei kannata siis poissulkea, jos tallennettavaa data on vähän. JSON-serialisaation tehokkuus XML-serialisaatioon

verrattuna saattaa selittyä sillä, että JSON-serialisaatioon käytettiin Unitylle optimoitua `JsonUtility`-luokkaa.

Käytännössä `Hero`-objektin tallennukseen ja lataukseen kuluu hyvin lyhyt aika, kun tallennettavaa dataa on vähän. Jos tallennettavaa dataa on vain muutaman peliobjektin verran, on tallennusprosessi niin lyhyt, ettei se ehdi vaikuttaa pelaajan pelikokemukseen negatiivisesti. Pienemmissä ja keskisuurissa peliprojekteissa ei siis välttämättä ole suurta merkitystä sillä, miten pelidatan tallentaa. Ohjelmoija voi valita itselleen kätevimmän tallennusmuodon ilman huolia siitä, että pelaajan pelikokemus kärsisi. Jos pelissä on kuitenkin huomattavasti tallennettavaa dataa, esimerkiksi tuhansia peliobjekteja tai monen megatavun edestä muuta tallennettavaa dataa, olisi järkevää panostaa tallennusjärjestelmän toiminnan optimointiin.

Mittaustulokset poikkeavat hieman saatavilla olevassa kirjallisuudessa esiintyvistä tuloksista. Yhdessä C#-kielen serialisaatiota tarkastellussa mittauksessa serialisoitavan datan määrän kasvaessa XML-tiedoston koko pieneni ja binääritiedoston koko kasvoi. Lisäksi binääridatan serialisaatioon ja deserilisaatioon kului mittauksissa enemmän aikaan kuin vastaavaan prosessiin kului XML-dataa käsiteltäessä. [17] Erot tämän tutkielman tuloksiin saattavat johtua siitä, että tämän tutkielman mittaukset suoritettiin Unity-pelimoottorin sisällä. Lisäksi pelkkää `Hero`-luokan repun kokoa muuttamalla ei välttämättä saa kunnollisesti simuloitua sellaista peliobjektia, joka sisältää paljon dataa.

4.6 Peliasetusten tallennus

Unityn tarjoaman `PlayerPrefs`-luokan avulla voi tallentaa merkkijonoja, liukulukuja sekä kokonaislukuja pelaajan järjestelmään. Tämä data säilyy pelisessioiden välillä. [26] `PlayerPrefs` on tarkoitettu pelaajan konfiguraation eli peliasetusten tallentamista varten [22]. Sitä voidaan esimerkiksi käyttää pelaajan ääniasetusten tallentamista varten. `Player-Prefs`in toiminnallisuutta voidaan kuitenkin käyttää myös pelidatan tallentamiseen. Ohjelmassa 4.9 tallennetaan `PlayerPrefs`iin äänenvoimakkuuden arvo avaimella `soundVolume`.

```
1 // Pelaaja muuttaa asetuksista äänenvoimakkuuden 80 prosenttiin
2 PlayerPrefs.SetFloat("soundVolume", 0.8f);
3 // Pelin käynnistyessä äänenvoimakkuuden arvo haetaan PlayerPrefsistä
4 audioPlayer.volume = PlayerPrefs.GetFloat("soundVolume", 1f);
```

Ohjelma 4.9. *Äänenvoimakkuuden tallennus ja lataus PlayerPrefsillä*

Tallennetun arvon voi myöhemmin hakea `PlayerPrefs`istä tämän avaimen avulla. Kun peli käynnistyessä asettaa äänilähteille äänenvoimakkuuden, peli yrittää etsiä äänenvoimakkuutta komennolla `PlayerPrefs.GetFloat` käyttäen avainta `soundVolume`. Jos tallennettu arvo löytyy, se asetetaan äänilähteiden äänenvoimakkuudeksi. Jos ei, äänenvoi-

makkuudeksi asetetaan `GetFloat`issa annettu oletusparametri, esimerkin funktiokutsussa `1f`.

4.7 Lisäosat

Unity Asset Store -verkkopalvelussa on saatavilla lukuisia yhteisön kehittämiä lisäosia, joiden avulla ohjelmoija voi liittää valmiin tallennusjärjestelmän projektiinsa. Esimerkiksi maksullinen *Easy Save*-lisäosa sisältää tuen lähes sadan yleisesti käytetyn muuttujatyypin automaattiseen tallennukseen serialisoinnin avulla [31]. Tarkastellaan, miten datan tallennus tapahtuu Easy Saven avulla.

Datan tallennusprosessi on ohjelmoijalle samanlainen kuin `PlayerPrefs`issä, mutta Easy Save tukee moninkertaisesti enemmän erilaisia muuttujatyyppejä kuin `PlayerPrefs` [31]. Easy Save tekee kaiken datan serialisoinnin ohjelmoijalta piilossa. Hero-olion tallennus Easy Saven avulla on esitetty ohjelmassa 4.10.

```
1 // HeroStatuksen tallennus
2 ES3.Save("HeroStatus", heroInstance.GetCurrentState());
3 // HeroStatuksen lataus
4 Hero.HeroStatus status = ES3.Load("HeroStatus", defaultValue);
5 heroInstance.LoadHeroState(status);
```

Ohjelma 4.10. *HeroStatuksen tallennus ja lataus Easy Save 3:lla*

Dataa tallennetaan antamalla Easy Saven metodille `ES3.Save` merkkijonomuotoinen avain sekä tallennettava data. Lataus tapahtuu antamalla metodille `ES3.Load` hakuavain sekä oletusarvo, jota käytetään jos tallennettua dataa ei löydy. [32]

5. YHTEENVETO

Valtaosa nykyajan videopeleistä sisältää järjestelmän, jonka avulla videopelin hetkellinen tila voidaan tallentaa ja myöhemmin palauttaa. Tallennusjärjestelmän avulla pelaaja pystyy keskeyttämään pelin ja myöhemmin jatkamaan samasta pelitilasta ilman, että peli olisi aloitettava kokonaan alusta. Tallennusjärjestelmän toiminnallisuuden kannalta on tärkeää, että pelitilan tallentamiseen ja palauttamiseen kuluu mahdollisimman vähän aikaa ja että tallennusprosessissa luodun tallennustiedoston koko olisi mahdollisimman pieni. Tähän voidaan vaikuttaa sekä itse pelimekaniikkojen suunnittelussa että tallennusjärjestelmän toteutuksella.

Tässä kandidaatintyössä tutkittiin tallennusjärjestelmien integrointia pelimaailmaan sekä tallennusjärjestelmän toteuttamismahdollisuuksia Unity-pelimoottorilla. Unity-pelimoottorilla videopelin tilan tallennukseen on järkevintä käyttää serialisaatiota. Dataa voidaan serialisoida erilaisiin tiedostomuotoihin. Serialisoitu data voidaan palauttaa alkuperäiseen muotoonsa deserialisoimalla tiedosto. Tässä työssä tarkasteltiin peliobjektin tilan tallentamista binääri-, XML- sekä JSON-muotoisiin tiedostoihin sekä datan lataamista luoduista tiedostoista.

Työssä luotiin Unity-pelimoottorilla yksinkertainen peliobjekti `Hero`, jolla oli nimi, tietty määrä elämäpisteitä, reppu joka sisälsi esineitä sekä tieto siitä, onko peliobjektin lähellä vihollisia. `Hero`sta tallennettiin vain ne tiedot, jotka olivat pelitilan säilymisen kannalta kriittisiä. Nimi, elämäpisteet ja repun sisältö oli pakko tallentaa. Tieto lähellä olevista vihollisista ei ollut pakollista tallentaa, sillä se on mahdollista päätellä pelitilasta pelin käynnistyessä. `Hero`-olio tallennettiin ja ladattin esimerkeissä binääritiedostoon, XML-tiedostoon sekä JSON-tiedostoon. Eri tiedostomuotojen tallennus- ja latausnopeutta sekä tiedostokokoa vertailtiin.

Binäärimuotoiset tiedostojen tallennukseen ja lataukseen kului testeissä vähiten aikaa. Tallennustiedoston tiedostokoko oli alhaisin, kun `Hero`-olio serialisoitiin JSON-muotoon. Datan tallennukseen ja lataukseen kulunut aika oli kuitenkin kaikilla tiedostotyypeillä mittauksissa niin lyhyt, ettei tallennustiedoston muodolla ole pelaajan pelikokemukseen merkittävää vaikutusta. Eri tiedostotyyppien väliset erot korostuvat vasta, kun tallennettavaa dataa on useita megatavuja. Pelin kehittäjät voivat siis melko vapaasti valita mieleisensä tallennustiedostomuodon ilman huolta siitä, että pelin suorituskyky tai pelikokemus kär-

sisivät. Jos pelissä kuitenkin on hyvin paljon tallennettavaa dataa, esimerkiksi tuhansia peliobjekteja, on järkevää panostaa tallennusjärjestelmän optimointiin.

Mittaustulokset erosivat erään aiemman C#-kielellä toteutetun serialisointimenetelmien vertailun tuloksista. Ero saattaa selittyä sillä, että tässä työssä datan serialisointi toteutettiin Unity-pelimoottorin sisällä. Lisäksi `Hero`-peliobjektin tietorakenne on erilainen kuin aiemmassa vertailussa.

Pelaajakohtaiset asetukset eli pelin konfiguraatio voidaan tallentaa käyttämällä Unityn PlayerPrefs-luokkaa. PlayerPrefsiin voisi tallentaa esimerkiksi pelimusiikin äänenvoimakkuuden arvon. PlayerPrefs ei kuitenkaan sovellu varsinaisen pelitilan tallentamiseen, vaan siihen suositellaan sen sijaan serialisaation käyttöä.

Unitylle on saatavilla useita lisäosia, jotka helpottavat tallennusjärjestelmän integrointia peliin. Tässä työssä tarkasteltiin Unityn lisäosakaupassa myytävän Easy Save 3-järjestelmän toiminnallisuutta. Järjestelmää käyttävän ohjelmoijan ei tarvitse perehtyä tarkemmin tallennusprosessissa tapahtuvaan serialisaatioon ja deserialisaatioon, mutta hänen on silti mietittävä, miten hän rajaa pelimaailmasta tallennettavan datan. Peliobjekteja voi tallentaa ja ladata vain yhdellä rivillä koodia järjestelmän avulla.

LÄHTEET

[1] Kenlon, S. Save Files and Game States. *Developing Games on the Raspberry Pi*. Berkeley, CA: Apress, 2018, s. 189–210. ISBN: 148424169X.

[2] Young, S. *Why We Have Checkpoint Saves*. URL: https://v1.escapistmagazine.com/articles/view/video-games/columns/experienced-points/10301-Why-We-Have-Checkpoint-Saves (viitattu 09. 03. 2021).

[3] *How to make a save system for your game (part 1)*. URL: https://karboosx.net/blog/post/how-make-save-system-your-game (viitattu 09. 03. 2021).

[4] Kalogiros, P. *Protecting and Cracking Game Save files*. URL: https://medium.com/@pantelis/protecting-game-saves-and-the-case-of-unworthy-e24c8fd68e16 (viitattu 09. 03. 2021).

[5] Christian, M. *A brief history of saved games*. URL: https://writing.markchristian.org/2019/02/02/saved-games/ (viitattu 09. 03. 2021).

[6] Savillo, R. *Game Design: Save Systems*. URL: https://venturebeat.com/community/2009/10/01/game-design-save-systems/ (viitattu 09. 03. 2021).

[7] Cooper, D. *How Kingdom Come: Deliverance Will Fix Its Save System*. URL: https://gamerant.com/kingdom-come-deliverance-save-fix/ (viitattu 09. 03. 2021).

[8] Adams, E. *Fundamentals of Game Design, Third Edition*. 2013.

[9] Christian, M. *Saved, But Not Forgotten*. 2019. URL: https://tedium.co/2019/02/21/video-game-save-state-history/ (viitattu 03. 03. 2021).

[10] Gurwin, G. *The best save points in video games*. URL: https://finance.yahoo.com/news/best-save-points-video-games-131501300.html? (viitattu 09. 03. 2021).

[11] Sirlin, D. Saving the day: save systems in games. *Game developer* 14.8 (2007). ISSN: 1073-922X.

[12] Kuchera, B. *The passion of the checkpoint: Why gaming's most frustrating failure is so hard to fix*. URL: https://www.polygon.com/2014/2/25/5422328/the-passion-of-the-checkpoint-why-gamings-most-frustrating-failure-is (viitattu 09. 03. 2021).

[13] Moran, C. Playing with Game Time: Auto-Saves and Undoing Despite the 'Magic Circle'. eng. *Fibreculture journal* 16 (2010). ISSN: 1449-1443.

[14] *How do I save and load a game using the quicksave/quickload options in Morrowind?* URL: https://help.bethesda.net/app/answers/detail/a_id/17355/kw/quicksave (viitattu 09. 03. 2021).

[15] *Binary File Definition, The Linux Information Project*. URL: http://www.linfo.org/binary_file.html (viitattu 16. 04. 2021).

[16] Sutherland, B. Using File IO to Save and Load Games. *C++ Game Development Primer*. Berkeley, CA: Apress, 2014, s. 33–48. ISBN: 9781484208151.

[17] Asad, A. ja Ali, H. Serialization and Deserialization. *The C Programmer's Study Guide (MCSD)*. Berkeley, CA: Apress, 2017, s. 305–318. ISBN: 1484228596.

[18] *Serialization in .NET*. URL: https://docs.microsoft.com/en-us/dotnet/standard/serialization/ (viitattu 09. 03. 2021).

[19] Gregory, J. What is a game engine?: *Game Engine Architecture, Third Edition*. Milton: A K Peters/CRC Press, 2019. ISBN: 1138035459.

[20] Gregory, J. Unity. *Game Engine Architecture, Third Edition*. Milton: A K Peters/CRC Press, 2019. ISBN: 1138035459.

[21] *C language specification, Ecma international*. URL: https://www.ecma-international.org/publications-and-standards/standards/ecma-334/ (viitattu 16. 04. 2021).

[22] Uccello, A. *How to Save and Load a Game in Unity*. URL: https://www.raywenderlich.com/418-how-to-save-and-load-a-game-in-unity (viitattu 10. 03. 2021).

[23] *JSON Serialization, Unity documentation*. URL: https://docs.unity3d.com/Manual/JSONSerialization.html (viitattu 11. 03. 2021).

[24] *XML and SOAP serialization*. URL: https://docs.microsoft.com/en-us/dotnet/standard/serialization/xml-and-soap-serialization (viitattu 11. 03. 2021).

[25] *XML Serialization in Unity*. URL: https://oguzkonya.com/xml-serialization-and-deserialization-in-unity/ (viitattu 11. 03. 2021).

[26] *PlayerPrefs, Unity documentation*. URL: https://docs.unity3d.com/ScriptReference/PlayerPrefs.html (viitattu 09. 03. 2021).

[27] *Important Classes - GameObject, Unity documentation*. URL: https://docs.unity3d.com/Manual/class-GameObject.html (viitattu 16. 04. 2021).

[28] *Vector3, Unity documentation*. URL: https://docs.unity3d.com/ScriptReference/Vector3.html (viitattu 13. 03. 2021).

[29] *Transform, Unity documentation*. URL: https://docs.unity3d.com/ScriptReference/Transform.html (viitattu 16. 03. 2021).

[30] *Serializable, Unity documentation*. URL: https://docs.unity3d.com/ScriptReference/Serializable.html (viitattu 10. 03. 2021).

[31] *Supported types, Easy Save 3 documentation*. URL: https://docs.moodkie.com/easy-save-3/es3-guides/es3-supported-types/ (viitattu 09. 03. 2021).

[32] *Getting Started, Easy Save 3 documentation*. URL: https://docs.moodkie.com/easy-save-3/getting-started/ (viitattu 19. 03. 2021).