

Unity Project: Save & Load system

15.9.2020

Päivän aihe

- Pelin datan ja tilan tallennus ja lataus / Save & Load system
 - PlayerPrefs
 - Serialisointi taktiikka
 - Pilvipalvelusta ja palvelimelta
- Yksitellen keskustellaan projekti edistymisestä
- Projektityöskentelyä

Pelin tallennus

Peli data ja muu tieto voidaan tallentaa monella eri tavalla. Jotte peliä voidaan pelata myös myöhemmin samasta kohtaa kuin aikaisemmin, samoilla tavaroilla ja etenemisellä, täytyy pelille luoda oma Save & Load systeemi jotta tämä voisi onnistua.

Jokainen peli haluaa tallentaa ja lataa datan ja muut tiedot eri tavoilla. Online pelillä täytyy olla omat tietokannat pelaajien datan hallinnalle, mutta lokaali peli voi tallentaa tiedot suoraan käyttäjän omalle koneelle.

Tietojen tallentamiseen ja lataamiseen voidaan käyttää Unity sisäistä PlayerPrefs ominaisuutta, serialisoida data omalle tiedostolle tai hakea ja tallentaa data pilvipalveluun tai omalle palvelimelle. Muitakin tapoja on, esimerkiksi XML & JSON, mutta ne eivät ole yhtä suojattuja kuin muut.

PlayerPrefs

PlayerPrefs on Unity sisäinen datan tallennus ominaisuus, jonka tarkoituksena on tallentaa ja käyttää pelaajan asetuksia ja tietoja pelisessioiden välillä.

PlayerPrefs on todella helppo tapa tallentaa pelaajan eteneminen pelissä ja myös muu data. PlayerPrefs kannattaa käyttää pelaajan asetusten tallentamiseen.

PlayerPrefs esimerkki:

`PlayerPrefs.SetInt("level", 2);` ⇒ Tallentaa pelaajan levelin muistiin, joka on arvoltaan "2"

`PlayerPrefs.Save();` ⇒ Tallentaa PlayerPrefs muutokset

`int currentLevel = PlayerPrefs.GetInt("level");` ⇒ Hakee pelaajan levelin ja asettaa sen "currentLevel" muuttujalle

PlayerPrefs heikkoudet

PlayerPrefsillä on omat heikkoudet, se ei ole salattu tai suojeltu. Sitä ei ole suunniteltu tallentamaan laajia määriä dataa. Isoin heikkous PlayerPrefsille on se, että sillä voidaan tallentaa helpoiten VAIN int, string ja float muuttujia. Sillä voidaan myös tallentaa Bool tai muita muuttujia mutta se on pakko tehdä käyttäen int, float tai string muuttujia.

Tämän takia, Position tai Rotation tallentaminen on hankalampaa, sillä ei voida suoraan tallentaa Vector3 tai Quaternion arvoja, se täytyy tehdä muodossa:

SetFloat("positionx", value) ⇒	GetFloat("positionx")
SetFloat("positiony", value) ⇒	GetFloat("positiony")
SetFloat("positionz", value) ⇒	GetFloat("positionz")

Serialisointi

Serialisoimalla (Serialize) tavalla voidaan tallentaa data suoraan uuteen tiedostoon. Tällä tavalla voidaan binäärisesti tallentaa data ja on luotettavampi ja turvallisempi kuin PlayerPrefs. Serialisoimalla voidaan tallentaa paljon laajemmin muuttujia ja omia classejä, mihin taas PlayerPrefs ei kykene.

Jotta Serialisointi taval saa käyttöön, tarvii Scriptiin lisätä pari “using” riviä:

```
using System;  
using System.Runtime.Serialization.Formatters.Binary;  
using System.IO;
```

Serialisointi



Ennen kuin datan pystyy tallentamaan, tulee data serialisoida. Tämän pystyy kätevästi tekemään luomalla scriptiin uuden luokan, joka ei periydy MonoBehaviouristä.

Esimerkki luokka:

```
[Serializable]
class SaveData
{
    public int savedInt;
    public string savedString;
    public bool savedBool;
}
```

Serialisointi

Nyt kun on luokka luotu, voidaan tallentaa data uuteen tiedoostoon.

```
void SaveGame()  
{  
    BinaryFormatter bf = new BinaryFormatter(); => Serialisoi datan  
    FileStream file = File.Create(Application.persistentDataPath  
        + "/MySaveData.dat"); ==> luo uuden tiedoston ja lukee sitä  
    SaveData data = new SaveData(); => Luodaan uusi SaveData (luokka joka luotiin)  
    data.savedInt = intToSave; => Muokataan luokan arvot niihin, halutaan tallentaa  
    data.savedString = stringToSave;   
    data.savedBool = boolToSave;   
    bf.Serialize(file, data); => Serialisoidaan data tiedostoon, eli tallennetaan  
    file.Close(); => Suljetaan tiedoston lukeminen  
    Debug.Log("Game data saved!");  
}
```


Serialisointi

Pelin datan hakeminen on samanlaista

```
void LoadGame(){
    if (File.Exists(Application.persistentDataPath + "/MySaveData.dat")) ⇒ Tarkistetaan onko tiedostoa olemassa
    {
        BinaryFormatter bf = new BinaryFormatter();
        FileStream file =
            File.Open(Application.persistentDataPath
                + "/MySaveData.dat", FileMode.Open); ⇒ Luetaan tiedostoa
        SaveData data = (SaveData)bf.Deserialize(file); ⇒ Deserialisoidaan tiedosto
        file.Close(); ⇒ Suljetaan tiedoston lukeminen
        intToSave = data.savedInt; ⇒ Asetetaan haetut arvot uuteen SaveData luokkaan
        stringToSave = data.savedString; ⇒ Asetetaan haetut arvot uuteen SaveData luokkaan
        boolToSave = data.savedBool; ⇒ Asetetaan haetut arvot uuteen SaveData luokkaan
        Debug.Log("Game data loaded!");
    }
}
```

Serialisointi

Pelin datan hakeminen on samanlaista

```
void LoadGame(){
    if (File.Exists(Application.persistentDataPath + "/MySaveData.dat")) ⇒ Tarkistetaan onko tiedostoa olemassa
    {
        BinaryFormatter bf = new BinaryFormatter();
        FileStream file =
            File.Open(Application.persistentDataPath
                + "/MySaveData.dat", FileMode.Open); ⇒ Luetaan tiedostoa
        SaveData data = (SaveData)bf.Deserialize(file); ⇒ Deserialisoidaan tiedosto
        file.Close(); ⇒ Suljetaan tiedoston lukeminen
        intToSave = data.savedInt; ⇒ Asetetaan haetut arvot uuteen SaveData luokkaan
        stringToSave = data.savedString; ⇒ Asetetaan haetut arvot uuteen SaveData luokkaan
        boolToSave = data.savedBool; ⇒ Asetetaan haetut arvot uuteen SaveData luokkaan
        Debug.Log("Game data loaded!");
    }
}
```