Contents

Additional Documentation

- Binary Serialization:

https://docs.microsoft.com/en-us/dotnet/standard/serialization/binary-serialization

- PlayerPrefs:

https://docs.unity3d.com/ScriptReference/PlayerPrefs.html

- JSON Serialization:

https://docs.unity3d.com/Manual/JSONSerialization.html

- JSON Tutorial

https://unity3d.com/learn/tutorials/topics/scripting/intro-part-two

- StreamingAssets Folder

https://docs.unity3d.com/Manual/StreamingAssets.html

Demo

Import the asset and run **Example_SaveLoad** and **Example_CustomFile** scenes for features demonstration.

I tested this on Windows, Android and WebGL. Theoretically, it should work on Mac and Linux as well, but I cannot test on those OS.

If you want to understand more about the inner workings follow to the next session, otherwise jump to “How to use it?” to know how to use this asset.

How does it work?

Depending on the platform is running on, the code will handle save/loading accordingly. First, it will check if the file exists, then creates a filestream to read/write. Lastly, the filestream is serialized/deserialized for write/read respectively.

The SaveSystem

What is Binary Serialization?

It is a process for storing an object’s state. During the process, the information is converted to a stream of bytes and written on a data stream. When deserializing, the opposite happens.

For more information on **Binary Serialization**, see the documentation on:
https://docs.microsoft.com/en-us/dotnet/standard/serialization/binary-serialization

Why Binary Serialization?

There are many ways for storing session data of games on Unity. For very simple stuff such as integers or text, Unity’s **PlayerPrefs** will suffice. However, if you want to store things like lists or structs it will not be possible. Another detail is that the information saved on PlayerPrefs are easily accessible, so is advisable to reserve it for game settings. (Sound volume, fullscreen…)

For more information on PlayerPrefs, see the documentation on:
https://docs.unity3d.com/ScriptReference/PlayerPrefs.html

Another alternative is the usage of **JSON** Serialization. JSON is a rather easy way to store data and you can even store structs and other things, making it a better alternative than PlayerPrefs for more complex data.

Note that any text editor can read and modify JSON files, at least on desktop. Therefore, if you do not want that the user should be able to edit stored information take this into consideration.
For more information on JSON serialization, see the documentation on:
https://docs.unity3d.com/Manual/JSONSerialization.html
Unity has a good tutorial series about JSON as well:
https://unity3d.com/learn/tutorials/topics/scripting/intro-part-two

With **binary serialization**, the information is still visible but altering it will render the file unusable, forcing the player to revalidate the installation to recover the correct files.

In the end, all methods are valid and broadly used. I was working on a project that required lots amount of data to be stored that should be hidden from the players. It was a narrative game and the storywriter did not want to be so easy to check for different paths or spoilers. Since we were using an auxiliary system to encrypt the story, which as pretty much a small book, we saved all text as blocks using binary serialization.

How to use it?

File Setup

Either edit GameData.cs or create a new class that derives from SaveFile.cs.
GameData.cs is a blank template ready to be used. You can add on it all kinds of variables that you want to be stored. For example, xp earned, level, stats, items found, achievements unlocked…
If you want to create a new class for saving data, it must derive from **SaveFile** and to be serializable. To do this add the attribute **[System.Serializable]** above the class declaration.
For example, see the code below:

```
1
2   namespace CrossPlatformSaveSystem
3   {
4       [System.Serializable]
5       public class SampleSaveFile : SaveFile
6       {
7           public int Clicks { get; set; }
8       }
9   }
10
```

The **SampleSaveFile** is used on the Example_SaveLoad scene. It stores how many times the button on the scene was clicked.

Saving and Loading Data

For simply saving or loading savefiles, I suggest using the **SaveManager.cs**. First, attach it to a gameobject on the scene you intend to deal if saved data. Then, on your script use **SaveManager.SaveDataToFile** when storing new data and **SaveManager.LoadDataFromFile** when loading data.

The methods are:

- Public static void **SaveDataToFile** (SaveFile **file**, string **fileName**)
- Public static SaveFile **LoadDataFromFile** (string **fileName**)

The script **Example.cs** demonstrates how this can be done.

First is declared an instance of the SampleSaveFile:

```
14          //Declares a instance of a SaveFile as SampleSaveFile
15          SampleSaveFile file;
```

When saving make sure that file is not null, otherwise it won't be saved properly. To avoid this, before calling **SaveManager.SaveDataToFile**, check if file is null and create a new instance if necessary.

```
60          public void ButtonSave (string filename)
61          {
62              //Create new instance of SampleSaveFile if file is empty
63              if (file == null)
64                  file = new SampleSaveFile ();
65
66              //Updates data of current file
67              file.Clicks = clicks;
68              //Calls SaveManager to save the file
69              SaveManager.SaveDataToFile (file, filename);
70
71              autoSave_display.text = "AutoSave file: " + filename;
72          }
```

Update the file with new data and call the function as above.

When loading, make sure to check if the returned file is not null. The reason to this is that **SaveManager.LoadDataFromFile** returns null if file with given name is nonexistent.

```
74      public void ButtonLoad (string filename)
75      {
76          //Loads file with given name
77          file = (SampleSaveFile)SaveManager.LoadDataFromFile (filename);
78
79          if (file == null)
80          {
81              autoSave_display.text = "AutoSave file: Null";
82          }
83          else
84          {
85              clicks = file.Clicks;
86              clicks_display.text = clicks.ToString ();
87
88              autoSave_display.text = "AutoSave file: " + SaveManager.CurrentSaveFile;
89          }
90      }
```

Managing multiple save files

Sometimes you want to give the player option of multiple save files, but during the game you need to keep tracking each one is being used. For this you might want to use the **AutoSave.cs**.

The scene Example_SaveLoad, comes with 3 slots and an autosave for demonstration.

To test it, first click some times on the button “Click Me!” and then choose one of the slots. The autosave will be automatically updated to the current save file.

When opening the scene again, the data will be automatically loaded from the autosave file.

To do this, simply call **SaveManager.SaveDataToAutoSave**, and the data will be stored in the current save file being used. If no save file was created earlier, the data won’t not be saved properly.

```
93      public void ButtonSaveToActiveFile ()
94      {
95          //Updates data of current file
96          file.Clicks = clicks;
97          //Calls SaveManager to save the file
98          SaveManager.SaveDataToAutoSave (file);
99      }
```

When loading, call **SaveManager.LoadDataFromAutoSave** and then check if is not null before using it. Again, if no save file was created, loading will return null.

```
101     public void ButtonLoadFromActiveFile ()
102     {
103         //Loads current file from autosave
104         file = (SampleSaveFile)SaveManager.LoadDataFromAutosave ();
105
106         if (file == null)
107         {
108             autoSave_display.text = "AutoSave file: Null";
109         }
110         else
111         {
112             clicks = file.Clicks;
113             clicks_display.text = clicks.ToString ();
114
115             autoSave_display.text = "AutoSave file: " + SaveManager.CurrentSaveFile;
116         }
117     }
118
```

The methods are:

- public static void **SaveDataToAutoSave** (SaveFile **file**)
- public static SaveFile **LoadDataFromAutoSave** ()

Using SaveSystem.cs directly

While SaveManager.cs is a bridge to the SaveSystem and other scripts, it's possible to call **SaveSystem.Save** and **SaveSystem.Load** directly.

Note that you need to make sure you are not saving/loading null SaveFiles. Doing this will generate problems at runtime of the type **NullReferenceException** when manipulating those instances.

The methods are:

- public static void **Save** (SaveFile **fileData**, string **fileName**)
- public static void **Save** (SaveFile **fileData**, string **fileName**, string **filePath**)
- public static SaveFile **Load** (string **fileName**)
- public static SaveFile **Load** (string **fileName**, string **filePath**)

Custom file and locations

It's possible to save files in a different folder, for example **StreamingAssets**. Files stored in the StreamingAssets folder will be accessible via a pathname. This means you can add files into the project to be loaded on release, independent on the platform.

For more information on StreamingAssets, see the documentation on:

https://docs.unity3d.com/Manual/StreamingAssets.html

Why custom files?

If for some reason you don't want to use scriptables objects you can store data that will be used on release for reference, but never altered by the user. In other words, on release build those files should be read-only.

In that narrative game I worked on, we stored all blocks of texts on the StreamingAssets folder. Those files had a ".blk" extension and could only be read from, never modified by user intervention as designed.

Testing

The scene **Example_CustomFile** demonstrates a way to have custom files saved on the StreamingAssets folder.

To test it, first play the scene and click on the button "Set". If does not exist yet, a StreamingAssets folder will be added to the project hierarchy. If it does not appear, try refreshing the hierarchy.

Additionally, three ".test" files will be created on StreamingAssets. Clicking on each choice button will load a different file and display the data, a string and 2 bools.

How to set a path and custom file?

On the script ExampleCustom.cs it is possible to see how to do this.

For this example, **SampleCustomFile** was used. It also derives from SaveFile just like GameData and SampleSaveFile, also has the attribute Serializable.

```
4       [System.Serializable]
5       public class SampleCustomFile : SaveFile
6       {
7           public int ID;
8           public string plot;
9           public bool trigger1;
10          public bool trigger2;
11      }
```

To create the custom file, first declare a new instance and set all variables that are necessary, as the example below:

```
50          //Creates a new instance of the SampleCustomFile
51          SampleCustomFile customFile = new SampleCustomFile
52          {
53              ID = index,
54              plot = plot,
55              trigger1 = trigger1,
56              trigger2 = trigger2
57          };
```

Then, use **SaveSystem.Save** to save the file with a given name and location:

```
65          //Save the newly created custom file at StreamingAssets folder
66          //You can add a subfolder in StreamingAssets for better organization
67          SaveSystem.Save (customFile, fileName, Application.streamingAssetsPath);
```

Different from save files, you need to specify the extension in the name. In this example, “SampleOption.test” was used.
The third parameter is file path, you can easily get StreamingAssets path by using **Application.streamingAssetsPath**, or use any other path you desire.

When loading, call **SaveSystem.Load** and make sure that the file is not null before using.

```
30          //Loads the custom file from StreamingAssets folder by given name
31          SampleCustomFile customFile = (SampleCustomFile)SaveSystem.Load (fileName, Application.streamingAssetsPath);
32
33          //If file exists, displays data
34          if (customFile != null)
35          {
36              //Display data from custom file
37              plot.text = customFile.plot;
38              trigger1.text = string.Format ("Trigger 1: {0}", customFile.trigger1);
39              trigger2.text = string.Format ("Trigger 2: {0}", customFile.trigger2);
40          }
```

Again, fileName should contain extension and don’t forget to give the path.

Questions, Suggestions, Feedback

Feel free to email me at justkrated@gmail.com.