Getting Started Guide

UnitySerializer is reasonably simple to use, but you have to grasp some basic concepts to use it well.

The idea of UnitySerializer is to give you an easy way to load and save progress in your game and to allow a player to continue from the current point at a later time.

You can also use UnitySerializer to create more complex scenarios such as saving the current state of a "Room" so that it continues from where it left of when a user re-enters it.

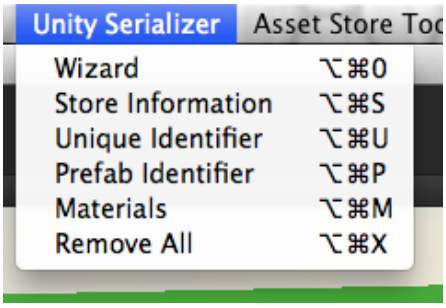
To start with we will consider the basic use of the system – providing a simple Save & Load feature.

Installation

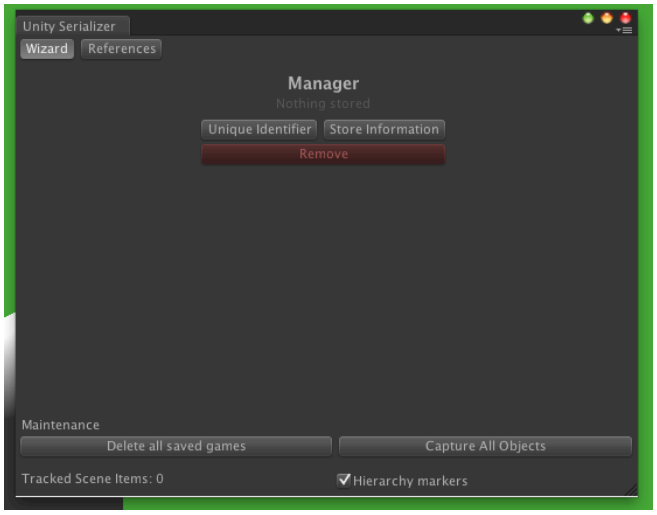
Install Unity Serializer by downloading the package from Asset Store or the web site and import it into your Unity project.

UnitySerializer Wizard

You can manually configure all of the features of UnitySerializer by adding scripts and setting properties, or you can use the UnitySerializer Wizard to automate much of the repetitive processes. This guide expects you to use the Wizard.



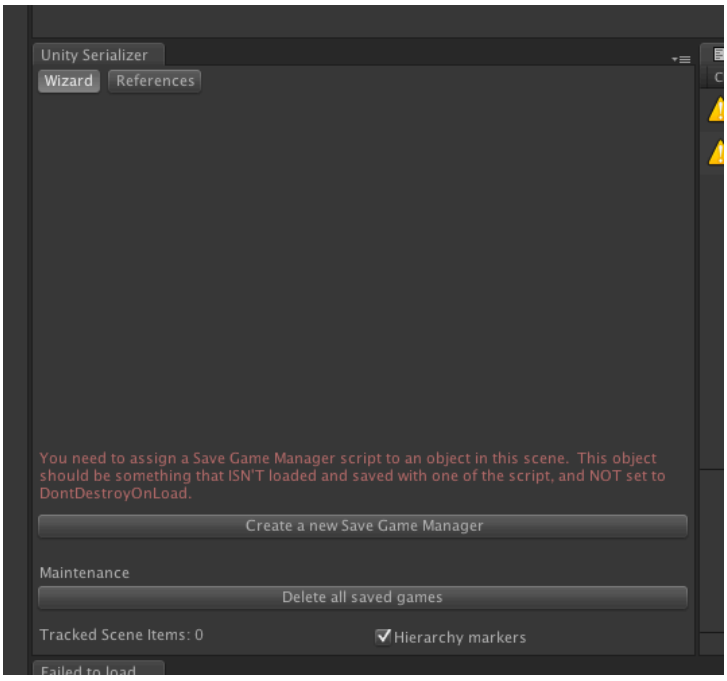
Having installed the package you will have a new top level menu called Unity Serializer – the wizard can be opened using this menu.



It is highly advisable to dock the wizard onto the side of the screen, because you will be frequently selecting different items and it can be frustrating to have to keep reopening the wizard.

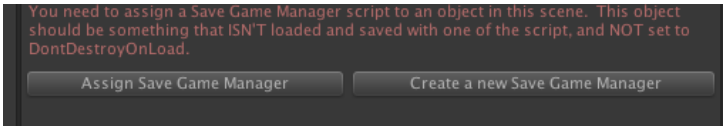
Preparing a scene

Firstly we need to prepare each scene in the game for saving. When you open a scene for the first time – the wizard will show a warning:



UnitySerializer warns you that you need to have a SaveGameManager in each scene.

The SaveGameManager is responsible for assigning a unique identifier to each object in the scene that has one of the storage scripts attached. It's a good idea to add one before adding any of the other scripts. Fortunately that is easy – all you have to do is click on the *Create a new Save Game Manager* button and a new object will be added to your scene. If you have an object selected in the scene you will see an extra button:



You can use the Assign button to make an existing object into the SaveGameManager – normally you will just create a new one however. If you do assign an existing object remember that the object that is a SaveGameManager must not be set to DontDestroyOnLoad and must not be saved and loaded with one of the Unity Serializer storage scripts.

Deciding what to save

The next step requires you to do a bit of thinking about your game. You are going to want to place one of the three Unity Serializer storage scripts on many of the objects in your scene and your project. It's up to you to work out which of the scripts are ideal for each scene object.

Start by knocking off the easy ones:

- Don't put any script on scenery; in other words, items that don't move, don't get destroyed and don't have any scripts (or only have scripts that don't need saving). This will be a lot of your scene including terrains etc.
- Prefabs – all of your prefabs that are *Instantiated* at run time must have a prefab identifier script attached.

Prefabs

The easiest way to put scripts on a prefab is to drop one into your scene temporarily and use the wizard to add a Prefab Identifier:

Click on the item – if the item is already a prefab you will see the *Prefab Identifier* button appear.

IMPORTANT: if your prefab has child objects then they *may* well need to be stored too. See the next sections on deciding what needs to be saved.

Storing information

Firstly, be aware that prefab identifiers store all the information on the scripts attached to the object on which they are located. Objects that are not prefabs can also be stored by using a Store Information script.

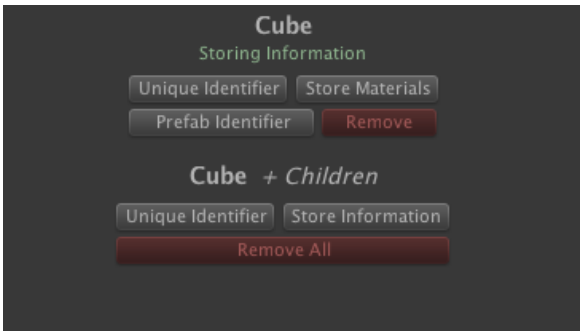
The easiest thing to do is to look at the rest of your scene objects and prefab children and ask yourself these questions:

1. Does this item move?
2. Is this item destroyed or changed?
3. Are scripts attached that have things that need to be saved? Examples would be player scores, enemy spawners, AI, game logic etc.
4. Does any other item refer to this item? That means the object or any script on it that you assign using the inspector or elsewhere in code.
5. Is this item the immediate parent of any item that is created while the game is playing?

If the answer to any of the first three questions is **yes** then you need a Store Information script. If all of the first three questions are answered **no**, but 4 or 5 are **yes** then you need a Unique Identifier script. You ***can*** just add a Store Information script if any of the questions is **yes** but your saved game will waste a bit of space if 1-3 are **no**.

Storing materials

By default materials are not saved because they don't often change and can take up a lot of space. If you do change material references or indeed create materials at run time procedurally you can choose to add a Store Materials script to any item that has a Prefab Identifier or a Store Information and the information about the materials will be saved along with the object. You can use the wizard to specify this or add the script manually.



Controlling what gets saved

Components

Each object with a Store Information or a Prefab Identifier script can choose which components and behaviours to save by using the Inspector. By default they store all of the components, and it's normal just to leave it that way.

Attributes

You don't need to worry about the attributes that are saved by standard Unity components – they are already sorted out for you. You do need to think about what will be saved on your own scripts – this is *very* important and if you don't take time over this part then things may very well throw errors or not quite give you the result you want.

By default all of the public fields and properties on your object will be saved. The private fields will not be saved. Often you might have important values held in private variables that you need to correctly restart the game from where it left off – if you do then you need to do one of the following:

- Flag private fields that you need with a *SerializeField* attribute which looks like this in JavaScript:

  @SerializeField
  private var myVariable : String;

  and like this in C#

  [SerializeField]
  private String myVariable;

- Store all of the private fields in addition to the public fields and properties by adding a *SerializeAll* attribute to your class like this in JavaScript:

  @script SerializeAll

  and like this in C#

  [SerializeAll]
  public class YourScript : MonoBehaviour

You also want to think "Are there things that shouldn't be saved?" if the answer is yes then you want to add a *DoNotSerialize* attribute in front of each of those. You can also add this to a class and it will never be serialized, even if variables that hold instances of it say they can be.

You can specify that a particular behaviour should not be saved by adding a *DontStore* attribute to the class.

What should I do?

Think about your scripts and work out which private variables are very important and make sure that they are saved. If it's most or all of them do this using a SerializeAll or otherwise mark each one with a SerializeField.

Skipping Initialization

Often your scripts will set up some default state in Awake or Start – you can normally just leave this unchanged, except if the that initialization would cause things to happen a bit later – like starting a coroutine, using Invoke or starting a fade on an animation.

If you have scripts that do this and you are loading information that would change what normally happened then you should skip the initialization if the level is being loaded.

That's very easy to do and is the same in C# and JavaScript:

```
if(LevelSerializer.IsDeserializing) return;
```

This is normally the first line of your Start, Awake or OnEnable when you have things that need skipping.

Fixing up after loading

More advanced scripts might need to do some work after loading to initialize some variables or look things up before the game recommences. This is relatively rare – but can prove to be a useful way of simplifying the number of things that need saving and can be very useful if accessing external resources.

Every script that is loaded can contain an OnDeserialized() function that will be called after the level has been loaded and before it starts to run. This function is exactly like the normal Unity functions and can be private or public as you wish. You would use it to do any final initialization.

For advanced users: Controlling what is loaded and saved

You can control what gets saved and loaded using a series of static events on LevelLoader and the ability to implement an interface on each of your scripts that controls whether a particular script will be saved. This allows you to have different modes of operation.

When loading data LevelLoader raised the following events:

CreateGameObject is triggered every time a game object is created from a prefab – you can cancel this operation of process the game object in some way.

OnDestroyObject is triggered when a scene object will be deleted.

LoadData is raised to tell you that data for a particular object will be loaded, you can cancel the load if you wish.

LoadComponent is raised when a specific component is being loaded – you can use this to cancel specific behaviours.

LoadedComponent is raised after a component has been loaded so that you can override behavior or in other ways modify the loaded item.

When saving data you can implement **IControlSerialization** on your scripts. This interface has one function **ShouldSave**() which returns *true* or *false*. False means no data will be saved for the component.

Loading and saving games

You work with LevelSerializer to load and save your games. LevelSerializer provides a number of ways of working with your game:

- A complete system for saving multiple games for each player
- A system for providing *checkpoint* and *resume* functions
- An API to get the saved data so you can put it anywhere you like, such as a server or a file

Example scripts

PauseMenu.js and TestSerialization.cs are two example scripts that show working with the complete system for saving and loading games.

Save & Load system

The easiest way to support level saving and loading is to use **LevelSerializer.SaveGame(**name**)**. Using this method you supply a suitable name for the game – this might change on different "levels" for instance, or might just be a single name all the time.

If you just call this method then a new entry will be made in PlayerPrefs for the saved game and the **LevelSerializer.SavedGames** collection will be updated. By default there are 20 slots per player.

To support multiple players you can set **LevelSerializer.PlayerName** to the screen name that represents the player – you can have as many of these as you like. By default PlayerName is blank and you can choose to leave it like this if you don't support the concept of different players.

To access the saved games you use the **LevelSerializer.SavedGames** static field – this is a lookup on PlayerName to a List of saved games.

Each of the entries in the list is a SaveEntry object that contains everything you need to display the entry, to load back the game and to delete the saved game.

Name is the name you provided for the game

When is the date and time the game was saved

Level is the name of the scene that was saved

Data is the actual data representing the saved game

Caption is a read only string that produces a suitable button caption for the saved game

Load loads the game

Delete deletes the saved game

Using checkpoints

It can be very handy to have checkpoints saved regularly or perhaps when the game is paused or goes into the background on a mobile device.

Calling **LevelSerializer.Checkpoint**() creates a new checkpoint entry with the players name (stored in PlayerPrefs).

Calling **LevelSerializer.Resume**() resumes from a previous checkpoint.

You can check LevelSerialize.CanResume() to see if there is data available for a resume.

> ***For advanced users: Storing data somewhere else***
>
> You can choose to store your level data anywhere you like by using the **LevelSerializer.SerializeLevel**() and **LevelSerializer.LoadSavedLevel**(data) functions.
>
> The LevelSerializer.SerializeLevel call returns a string that you can store. LevelSerilizer.LoadSavedLevel takes that string back and reloads the game.

> ***For advanced users: Partial saving***
>
> You might want more fine-grained control over what is saved or you might only want to save specific GameObjects and their children (there is an example of this in the OutOfRangeManager demonstration which automatically saves and destroys objects that are a far distance from the player).
>
> You can call a different version of LevelSerializer.SerializeLevel that takes two parameters. This first is a bool that says whether the game should be saved even if the serialization system is suspended (normally pass false) and the second is the ID of the object that will be the root of the saved data.
>
> This version returns a byte[] array and does not run standard compression on the data (for improved performance). You can compress the data yourself using:
>
> ```
> SevenZipRadical.Compression.LZMA.SevenZipRadicalHelper.Compress(data)
> ```
>
> And you can convert it to a string using Convert.ToBase64String()

You can reload the saved data using an instance of the LevelLoader component:

```
var l = new GameObject();
var loader = l.AddComponent<LevelLoader>();
loader.showGUI = false;  //Don't fade the screen to white
yield return new WaitForEndOfFrame();
//Load the data (example)
var f = File.Open(Application.persistentDataPath + "/" + item.id +
".dat", FileMode.Open);
var d = new byte[f.Length];
f.Read(d, 0, (int)f.Length);
f.Close();
yield return new WaitForEndOfFrame();

//Deserialize it
var ld = UnitySerializer.Deserialize<LevelSerializer.LevelData> (d);
loader.Data = ld; //Assign the data
loader.DontDelete = true; //Don't delete anything not in the file
//Get the loader to do its job
yield return loader.StartCoroutine(loader.Load());
```

Working with coroutines

You will probably want your coroutines to keep running from the point they left off.

To do this you need to do three things.

- Add a RadicalRoutineHelper to a game object in your scene and make sure that the game object is saved using Store Information.
- Start your coroutines by calling RadicalRoutineExtensions.StartExtendedCoroutine(object, routine)

  In C# you can use this as an extension method for Component, GameObject or MonoBehaviour – so

  ```
  var r = myGameObject.StartExtendedCoroutine(myCoroutine());
  ```

  In JavaScript you need to do it like this:

  ```
  var r = RadicalRoutineExtensions.StartExtendedCoroutine(gameObject,
  myCoroutine());
  ```

  The object returned by these methods can be used to cancel the coroutines execution.
- Yield any coroutines started by your coroutines using StartExtendedCoroutine as described above.

Getting Progress Notifications

You can handle the static event Progress on LevelSerializer to be informed regularly as your game is saved and loaded. The Progress event handler takes two parameters - a string indicating the section being processed and a float between 0 and 1 indicating how complete the save is.

Here is some example code for logging the progress, you would probably want to use some kind of graphic to indicate this to the player.

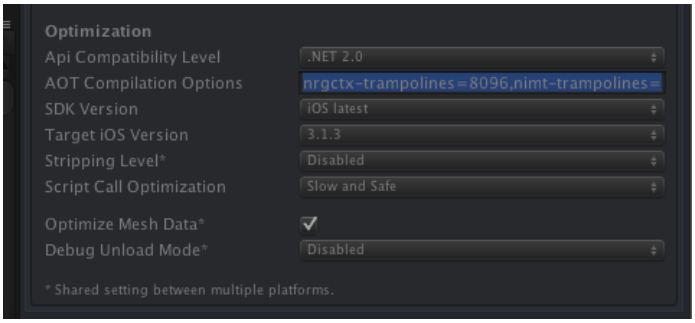
```
void OnEnable()
{
        LevelSerializer.Progress += HandleLevelSerializerProgress;
}

void OnDisable()
{
        LevelSerializer.Progress -= HandleLevelSerializerProgress;
}

static void HandleLevelSerializerProgress (string section, float complete)
{
        Debug.Log(string.Format("Progress on {0} = {1:0.00%}", section, complete));
}
```

iPhone and Android Considerations

Mono uses things called Trampolines in AOT (Ahead Of Time) compilation. These put quite severe limits on the use of interfaces and generics, which Unity Serializer uses a lot of. If you don't personally use these features then you will probably be fine, but if you do you may need to increase the budget available for them by setting some command line options in Player Settings.



Mine are set like this:

```
nrgctx-trampolines=8096,nimt-trampolines=8096,ntrampolines=4048
```

Explanation

nrgctx-trampolines=8096 (these are recursive generics – the default is 1024)

nimt-trampolines=8096 (these are to do with interfaces and the default is 128)

ntrampolines=4048 (are to do with generic method calls, by default there are 1024)