Component Save System Manual

1. Basics
    a. Creating your first saveable game object
    b. Writing your first saveable component
    c. Spawning saveable prefabs / instances of GameObjects
    d. Saving and loading your last played scene(s)
    e. Creating save screenshots and displaying them
    f. Configuring the save settings
    g. Storing simple data, similar to PlayerPrefs

2. Advanced
    a. Saving and loading your project manually
    b. Loading and writing metadata
    c. Accessing savegame classes directly

3. Components
    a. Component: SavePosition
    b. Component: SaveRotation
    c. Component: SaveScale
    d. Component: SaveVisibility
    e. Component: WriteSaveToDisk
    f. Component : Save Last Used Scenes
    g. Component : Load Last Used Scenes
    h. Component : Save Screenshot
    i. Component : Save Screenshot Displayer

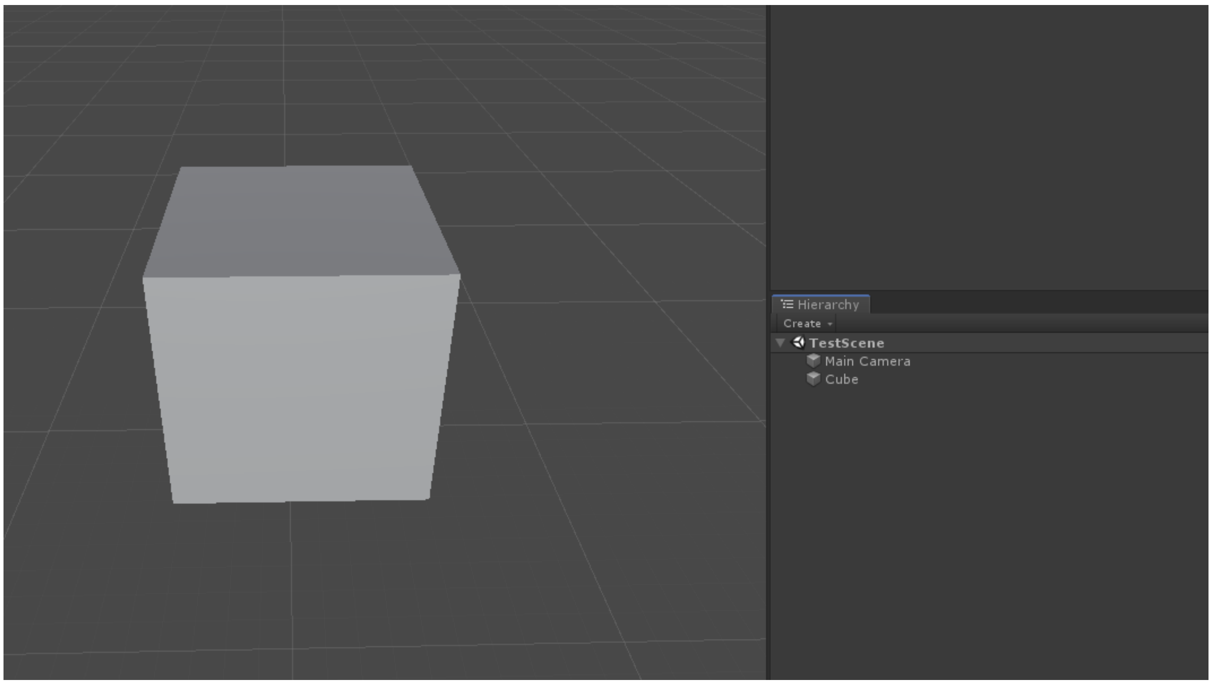
4. Examples
    a. Bomber Dog - Sample game
    b. Sample Main Menu
    c. Sample Pause Menu

5. About
    a. Contact
    b. Licence

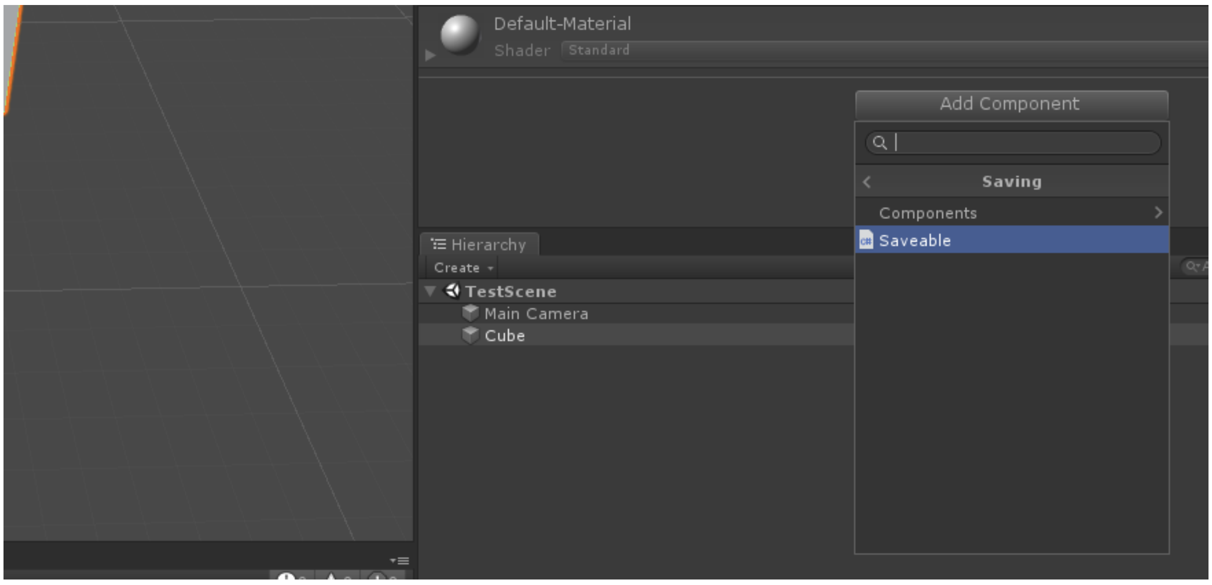
Creating your first saveable game object

By default the Save Component System automatically loads a save slot. And it automatically writes this to disk when the application closes. This means saving works right after you import the Save Component System.

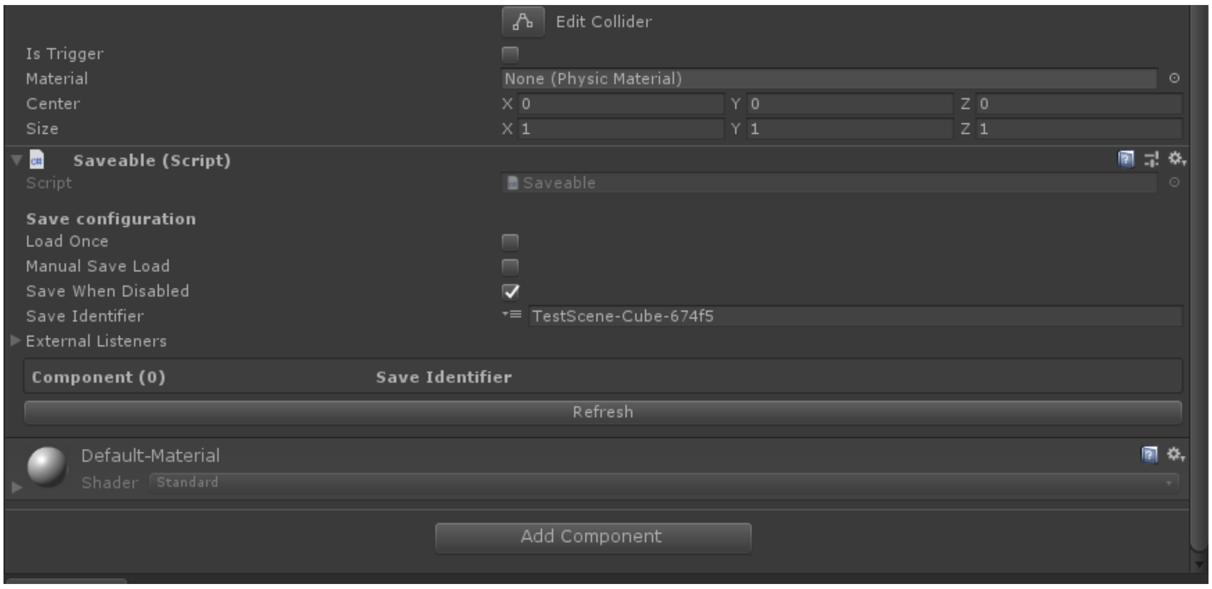
As an example, we create a scene with a simple cube.



After creating this cube nothing is saved yet. To ensure an object can save things we have to add a component called **Saveable.** You can find it by using the search function in the add component menu. Or by navigating to **Saving>Saveable** in the dropdown.



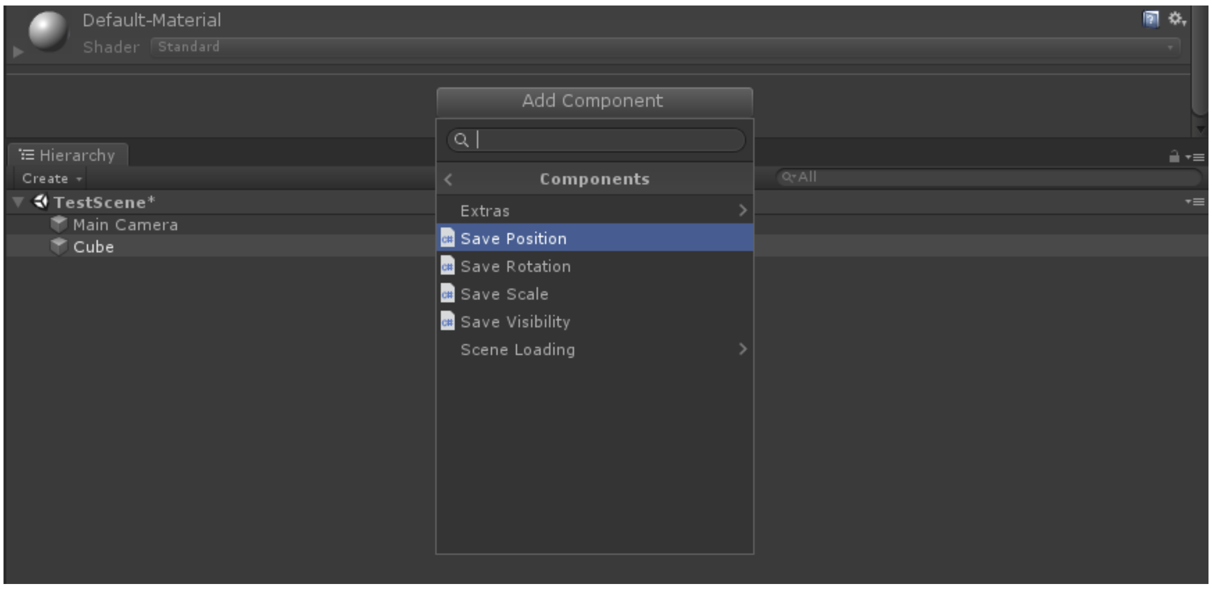
After adding the saveable component you can see that it contains several options. Most of these are relevant to specific niche use cases. The most important part of a saveable is the Save Identifier. The identifier is used to write an “Address” in a book, so to speak. By default the id that is generated is **<SceneName>-ObjectName-<RandomGUID>** however, it is not mandatory to keep this convention.



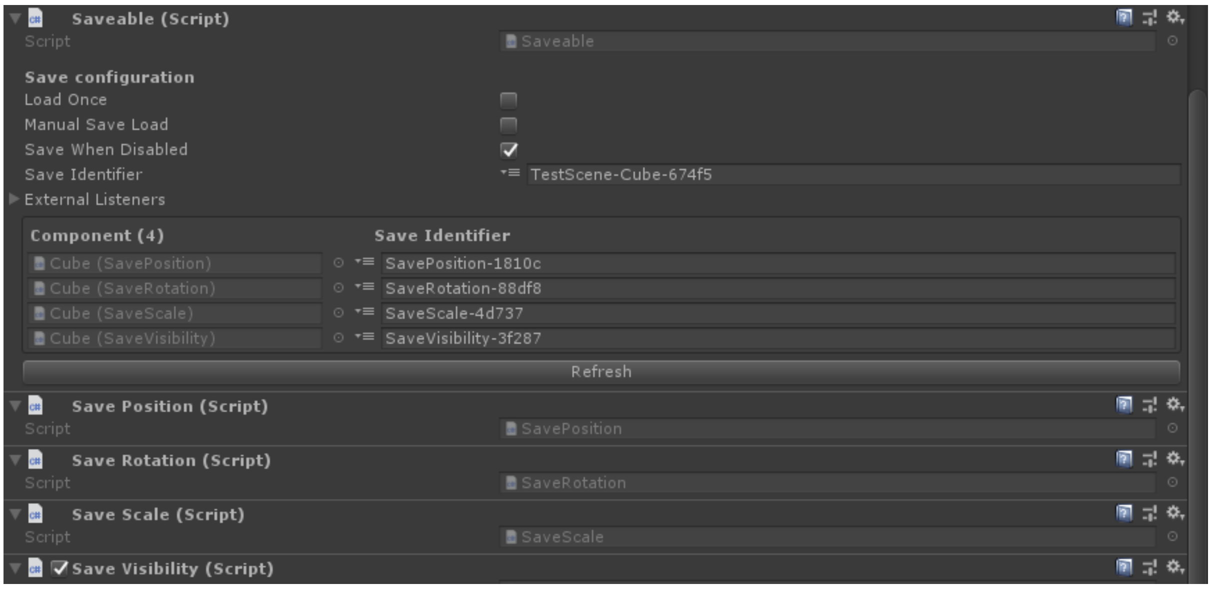
Is Trigger ☐
Material: None (Physic Material)
Center: X 0 Y 0 Z 0
Size: X 1 Y 1 Z 1
Script: Saveable
Save configuration
Load Once ☐
Manual Save Load ☐
Save When Disabled ✓
Save Identifier: TestScene-Cube-674f5
External Listeners
Shader: Standard

As you can see it also says **Component (0)** what this means, is that zero components will get saved by this Saveable. So no data will get written to the save game by this game Object, yet.

There are some premade components that make it easy for you to get up and running. You can find them by going to **Add Component > Saving > Components**



After adding several components you can see it automatically has created multiple identifiers. These are used to identify the component that will be saved to the save game.



Saveable (Script)

Script	Saveable

Save configuration

Load Once	☐
Manual Save Load	☐
Save When Disabled	✓
Save Identifier	TestScene-Cube-674f5

External Listeners

Component (4)	Save Identifier
Cube (SavePosition)	SavePosition-1810c
Cube (SaveRotation)	SaveRotation-88df8
Cube (SaveScale)	SaveScale-4d737
Cube (SaveVisibility)	SaveVisibility-3f287

Save Position (Script)
Script: SavePosition

Save Rotation (Script)
Script: SaveRotation

Save Scale (Script)
Script: SaveScale

Save Visibility (Script)

If you now run the game, move the cube, scale it and rotate it using the editor. And exit the game and load it back up again. You will see that it automatically applies the changes that happened previously. This is because the saveGame gets written to disk automatically upon game quit. Also, the saveable starts writing data to the save game when it gets destroyed, or when a sync request happens (?).

Writing your first saveable component

Unity uses components for mostly everything. The save system can benefit both very modular, and more monolithic frameworks. The component save system is meant to save components individually, without having to really think much about the context of the object. You can make a ton of saveable objects without having to write any custom lines of code.

In order to make a component saveable you have to implement an interface called ISaveable. In case you are not very familiar with interfaces yet. Don't worry, they are not as scary as you think. If you know about inheritance, for instance a dog inherits from animal. Then interfaces would be something that could be added more loosely to a class, sort of like a component. You ask a class to use specific method(s). Instead of it becoming part of a class like an animal, it rather has a characteristic. In this case, the characteristic is called **ISaveable**

First, before we add the interface to the class we need to include the namespace:

```
using Lowscope.Saving;
```

After including the namespace you can add the interface

```
public class ExampleScript : MonoBehaviour, ISaveable
{
}
```

The interface requires the following methods to be active inside your class

```
public void OnLoad(string data)
{

}

public string OnSave()
{
	return "";
}

public bool OnSaveCondition()
{
	return true;
}
```

If you want to check if a component has been loaded during Awake() or Start() you can now call **Saveable.IsComponentLoaded(this)** within a component that implements ISaveable.

Do note, that many tools like Visual Studio and Rider support automatic generation of code when you want to implement an interface. Making the process of making implementing them less tedious. For instance in Visual Studio you can press ALT+ENTER when hovering text.

To explain the methods, **OnLoad(string data)** returns you the data you have previously sent to the save game. ***This method is called during the Awake() step. Meaning you can already made use of any changes during the Start() method.***

The **OnSave()** gets called when a sync request is done, or when the game object is getting destroyed. This will allow you to write save data to the save game.

OnSaveCondition() is mainly as a performance measurement, making it easy to define if the object should get saved (this time) or not.

Now as a simple test, we can write a component that stores the name of the game object:

```
using Lowscope.Saving;
using UnityEngine;

public class ExampleScript : MonoBehaviour, ISaveable
{
	// Called before Awake()
	public void OnLoad(string data)
	{
		this.gameObject.name = data;
	}

	// Called when SyncSave is called or when saveable has been destroyed
	public string OnSave()
	{
		return this.gameObject.name;
	}

	// Used to check if it should save or not
	public bool OnSaveCondition()
	{
		return true;
	}
}
```

Now when you add this component to the object. You will see it will also get added to the list of saved components. And when you play the game, and change the object name and return. It will have the name saved!

You may think, why is this only returning strings? The main initial focus of this component was ease of use and flexibility. (Still quite performant, don’t worry)

The ease of use came through using the JSONUtility. This allows you to write a class full of data to a string, and convert it back to a class.

This example shows you how you could store character stats, and load them again.
Give it a go in the inspector! Do note, when making a class you have to add the [System.Serializable] attribute. Else it won't show up in the unity inspector, and also won't be serializable (convertible to JSON etc).

```
using Lowscope.Saving;
using UnityEngine;

public class Stats : MonoBehaviour, ISaveable
{
	[System.Serializable]
	public class StatData
	{
		public int activeHealth;
		public int totalHealth;
		public int strength;
		public int dexterity;
		public int intelligence;
	}

	[SerializeField] private StatData statData;

	public void OnLoad(string data)
	{
		statData = JsonUtility.FromJson<StatData>(data);
	}

	public string OnSave()
	{
		return JsonUtility.ToJson(statData);
	}

	public bool OnSaveCondition()
	{
		return true;
	}
}
```

You could also purely create classes for data. Then it often helps to just save it like this. And load the data accordingly.

```
return JsonUtility.ToJson(new StatData()
{
	activeHealth = this.activeHealth,
	dexterity = this.dexterity,
	intelligence = this.intelligence
});
```

Spawning saveable prefabs

Sometimes you need to spawn objects and need to store what has been spawned
Examples of these kind of objects could be, droppable items, bombs, fire.. etc

The simplest way to spawn a saveable prefab is by using the following method

```
SaveMaster.SpawnSavedPrefab(InstanceSource.Resources, "myPrefabName");
```

What this does is, it tries to spawn a prefab from the resources folder with the name myPrefabName. You can change this name to something you are familiar with.

For the best results I recommend adding a saveable component to an saveable prefab. This ensures all the fetching of ISaveables happens during runtime,

After the object is spawned, an instance manager will create an ID for the saveable and spawn it again using the parameters given initially.

Creating a custom resource loader

Since version 1.1 you are able to create your own custom resource loaders.
Doing so is quite straightforward.

```
SaveMaster.AddPrefabResourceLocation("ExampleCustomPrefabSpawner", LoadPrefab);
```

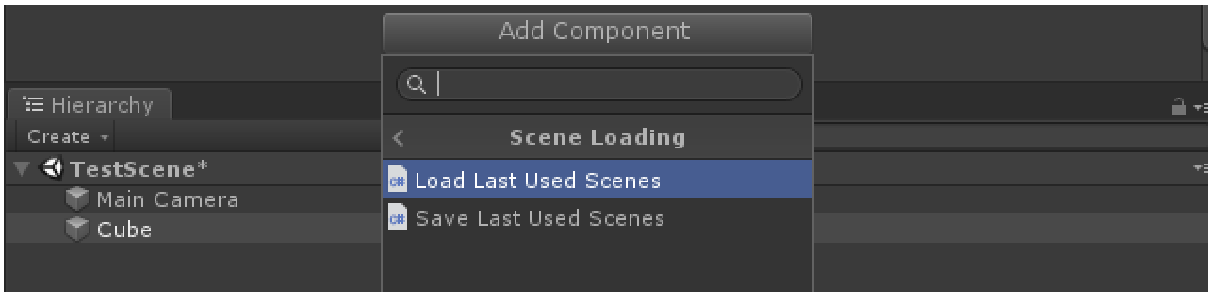
The LoadPrefab method has to be a function that returns a GameObject and uses a string as a parameter to define the index.

```
private GameObject LoadPrefab(string id)
{
        return myResource;
}
```

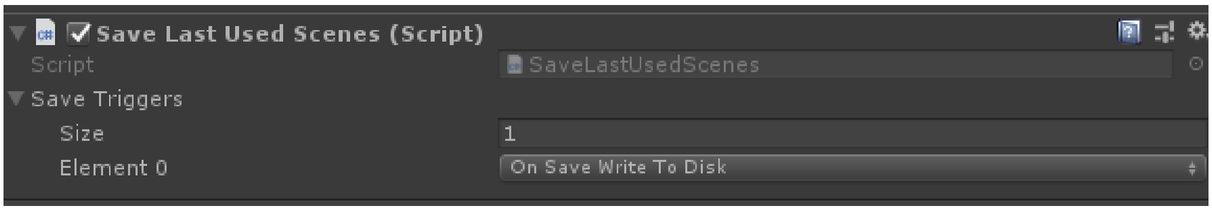
Within the LoadPrefab you can add code that returns a prefab instance. Currently there is no support for async loading of saveables. In theory this shouldn’t be advised anyway, because this can create differences upon each load due to load time differences. Personally i advise to do async loading within components attached on the saveable. Custom spawners may cope in a later version.

Automatic saving and loading last loaded scene(s)

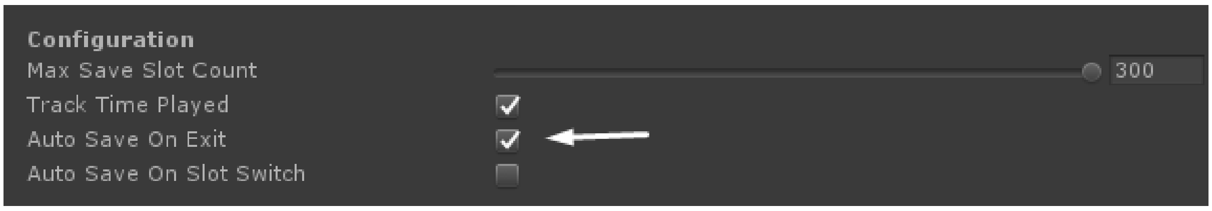
Since version 1.1 you are able to automatically load and save the last played scenes.
You can do this by using the LoadLastUsedScenes and SaveLastUsedScenes component.



Saving your active scene is as simple as adding the following component to one of your scenes. There are several options as triggers. On Save Write To Disk works when you decide to call **SaveMaster.WriteActiveSaveToDisk()** during gameplay. Or when you have the **Auto Save On Exit** or Auto Save On Slot Switch option turned on. **With the default settings this should just work and save the last used scenes.**

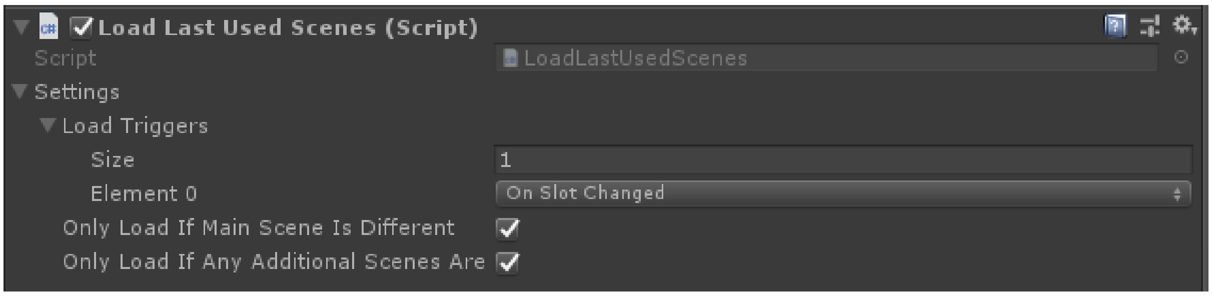


Save Last Used Scenes (Script)
Script: SaveLastUsedScenes
Save Triggers
Size: 1
Element 0: On Save Write To Disk



Configuration
Max Save Slot Count: 300
Track Time Played: ✓
Auto Save On Exit: ✓
Auto Save On Slot Switch: ☐

To load your scene you can use the **Load Last Used Scenes** component



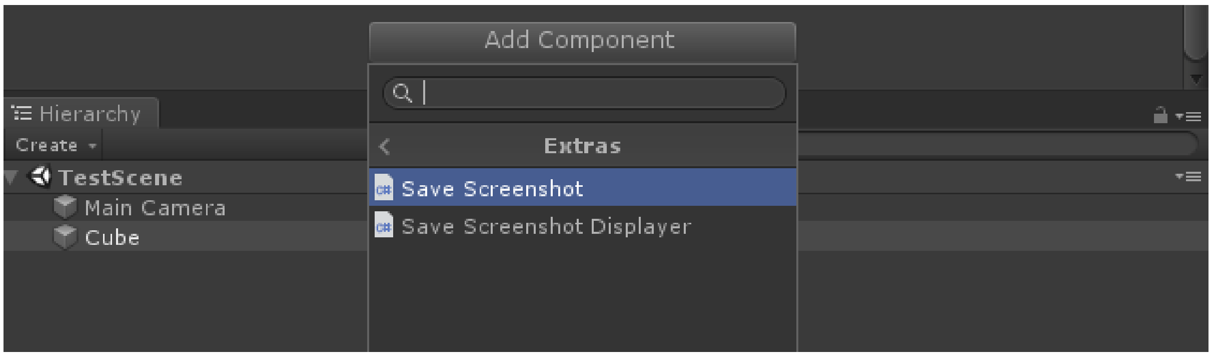
Load Last Used Scenes (Script)
Script: LoadLastUsedScenes
Settings
Load Triggers
Size: 1
Element 0: On Slot Changed
Only Load If Main Scene Is Different: ✓
Only Load If Any Additional Scenes Are: ✓

The default settings will load the last saved scene directly. It is possible to add both components to a prefab that is placed in all scenes to ensure the latest loaded scenes will always get loaded..

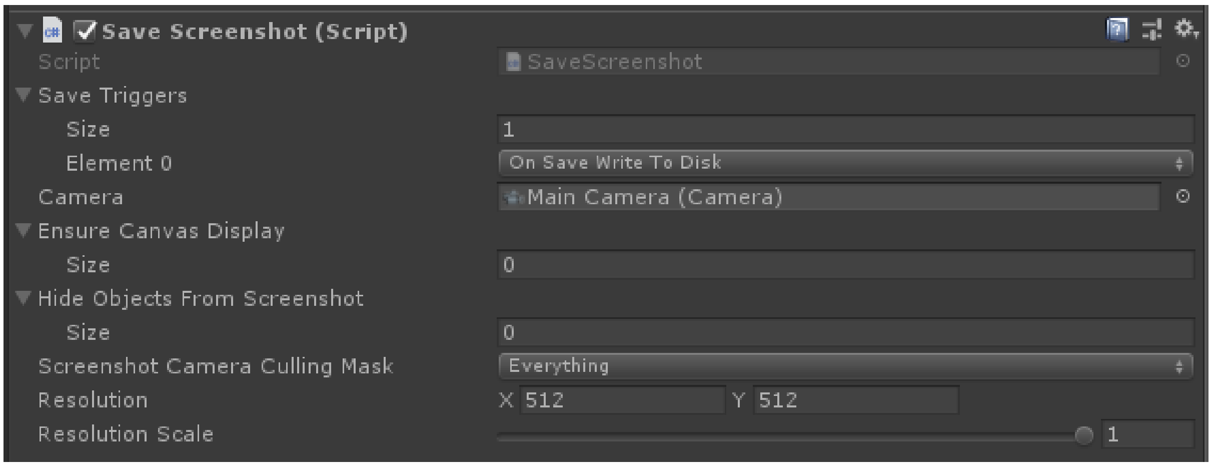
Creating save screenshots and displaying them

Creating screenshots of your game has been made a lot easier since version 1.1.
This version now supports separate metadata files that contain data about a save game.
Making it unnecessary to load each save game to fetch specific information about it.

You can find two components that make it easy for you to either, make a screenshot, or to display it on a Canvas object. You can find the components by going to
Add Component > Saving > Components > Extras.



Using the Save Screenshot component

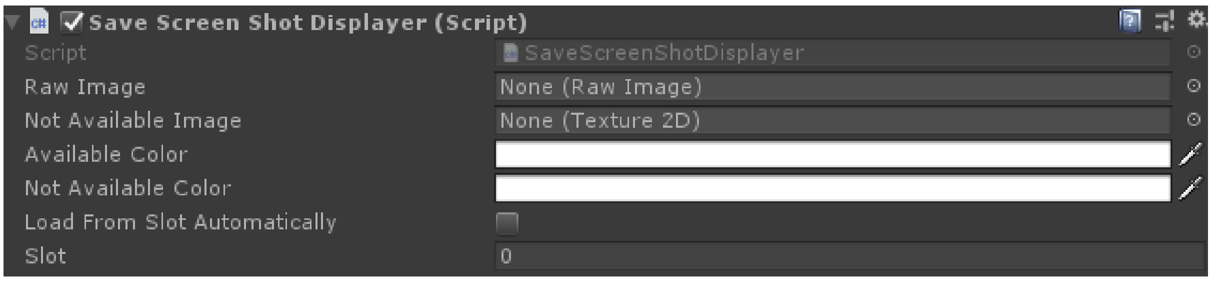

Save Screenshot (Script)

Script	SaveScreenshot
Save Triggers	
Size	1
Element 0	On Save Write To Disk
Camera	Main Camera (Camera)
Ensure Canvas Display	
Size	0
Hide Objects From Screenshot	
Size	0
Screenshot Camera Culling Mask	Everything
Resolution	X 512 Y 512
Resolution Scale	1

The simplest way to use the save screenshot component is to keep it at default settings. The biggest issue you may run into is that specific menus display during saving. You can use either the **Culling Mask** or use **Hide Objects from Screenshot** to fix this. These options revert the culling mask and objects to their previous state once the screenshot is made.

The Ensure Canvas Display option is to ensure that a specific canvas is not set to **Screen Space - Overlay**. Because this causes it to not get rendered on the screenshot.

Using the Save Screenshot Displayer



Save Screen Shot Displayer (Script)

Script	SaveScreenShotDisplayer
Raw Image	None (Raw Image)
Not Available Image	None (Texture 2D)
Available Color	
Not Available Color	
Load From Slot Automatically	☐
Slot	0

In order to make a Raw Image, you can go to the action bar at the top of Unity, select GameObject > UI > Raw Image. This will create an object that contains a raw image component. Reason we need this instead of a regular Image, is because a Texture2D reference has to be set, which is only supported by the Raw Image component.

If you are having issues with getting a proper aspect ratio. As in, the image is stretched in some kind of way. Then this is because the aspect ratio does not match the aspect ratio of the screenshot. The easiest way to match it is if you are making screenshots of an 1:1 aspect ratio. Such as 128x128, 256x256 or 512x512. Meaning you can just use a square shape for the UI element. If you are using a more custom resolution, a trick to make it easy is to run the game, and press **Set Native Size**. And then multiplying the width and height by 0.25 in the inspector to match what you want. (It's possible to do 1020*0.2 in the inspector, it will calculate for you)

The **not available image** is used if no image metadata has been found. Having this not set will make it display a plain color. Based on the available color and not available color.

Load from slot automatically is useful if you have a fixed amount of slots you want to show, and you do not want to do any programming for displaying this yourself. If toggled on, just ensure to change the **slot** value to the slot you want to display the image of. **The load is done automatically when the component is enabled.**

Refreshing the displayer from code

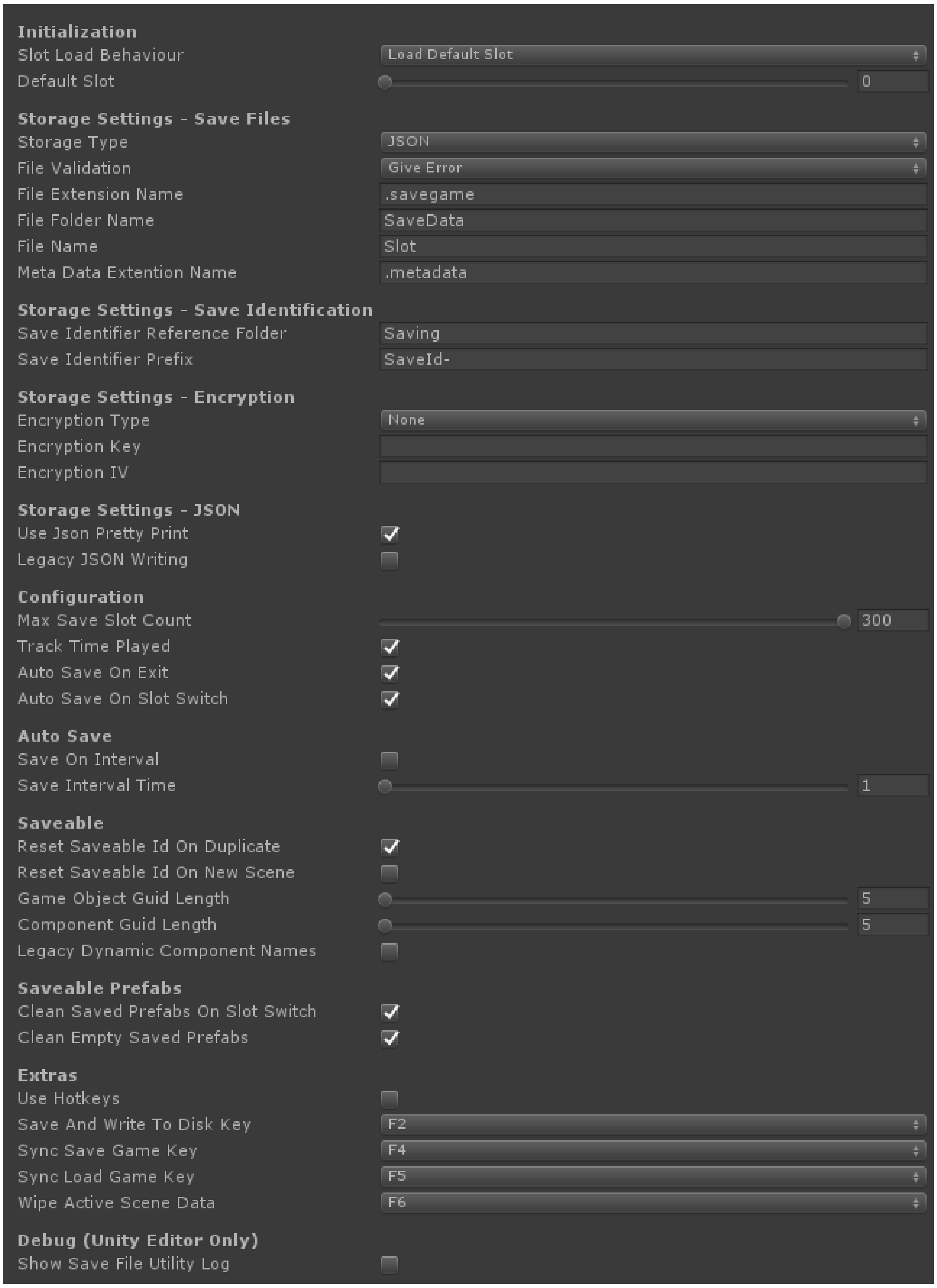
If you access the SaveScreenShotDisplayer from code, you can call the following method to display a saved image.

```
LoadScreenshot(int slot)
```

If you want a custom way to load the screenshots from the metadata, I recommend you take a look at the internals of this component. It is quite simple. However this component should cover most use cases.

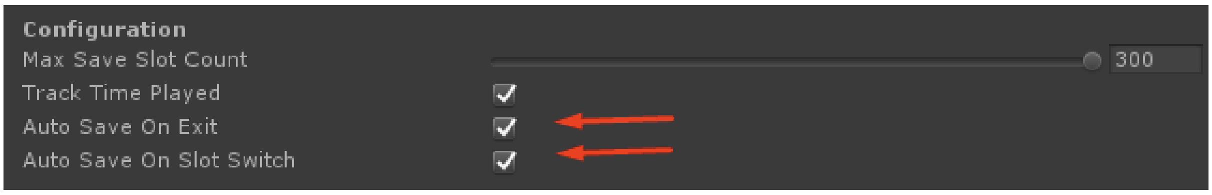
Configuring the save settings

In order to find this menu you have to go to **Window > Saving > Open Save Settings**
You can also find the settings file in your resources folder.



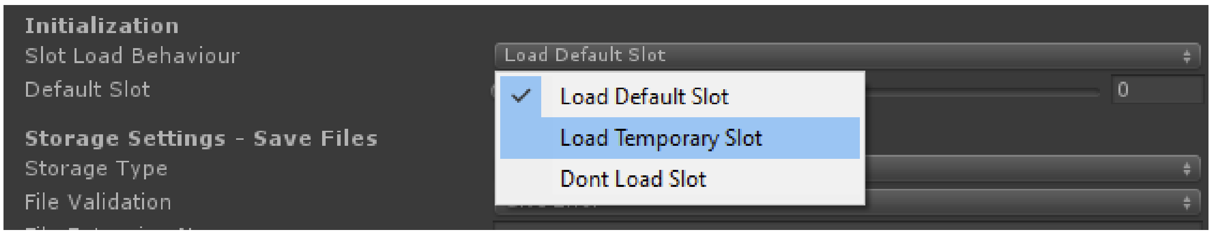
Initialization	
Slot Load Behaviour	Load Default Slot
Default Slot	0
Storage Settings - Save Files	
Storage Type	JSON
File Validation	Give Error
File Extension Name	.savegame
File Folder Name	SaveData
File Name	Slot
Meta Data Extention Name	.metadata
Storage Settings - Save Identification	
Save Identifier Reference Folder	Saving
Save Identifier Prefix	SaveId-
Storage Settings - Encryption	
Encryption Type	None
Encryption Key	
Encryption IV	
Storage Settings - JSON	
Use Json Pretty Print	✓
Legacy JSON Writing	☐
Configuration	
Max Save Slot Count	300
Track Time Played	✓
Auto Save On Exit	✓
Auto Save On Slot Switch	✓
Auto Save	
Save On Interval	☐
Save Interval Time	1
Saveable	
Reset Saveable Id On Duplicate	✓
Reset Saveable Id On New Scene	☐
Game Object Guid Length	5
Component Guid Length	5
Legacy Dynamic Component Names	☐
Saveable Prefabs	
Clean Saved Prefabs On Slot Switch	✓
Clean Empty Saved Prefabs	✓
Extras	
Use Hotkeys	☐
Save And Write To Disk Key	F2
Sync Save Game Key	F4
Sync Load Game Key	F5
Wipe Active Scene Data	F6
Debug (Unity Editor Only)	
Show Save File Utility Log	☐

Disabling automatic writing to disk on exit or slot change


Configuration
Max Save Slot Count 300
Track Time Played ✓
Auto Save On Exit ✓
Auto Save On Slot Switch ✓

In case you want a game that has no automatic saving. Or that uses specific locations as save points. It is recommended to turn off **Auto Save On Exit** and **Auto Save On Slot Switch**. This ensures nothing gets written to the disk automatically, no matter what slot you load.

Loading a temporary slot, overwriting other slots with current progress


Initialization
Slot Load Behaviour Load Default Slot
Default Slot 0

Storage Settings - Save Files
Storage Type
File Validation

This methodology is good for games like Doom, Elder Scrolls: Skyrim. Where you don't want to use any automatic saving. You don't want to load any progress when starting a game. And saving mostly consists of writing the active data to a particular slot. Loading the game should still happen in the regular way.

In order to use a temporary slot, you have to set the **Slot Load Behaviour** to **Load Temporary Slot**. What this does is it just loads an empty slot, but it will still keep data in memory when you change scenes. So you still switch scenes and have data stored in memory.

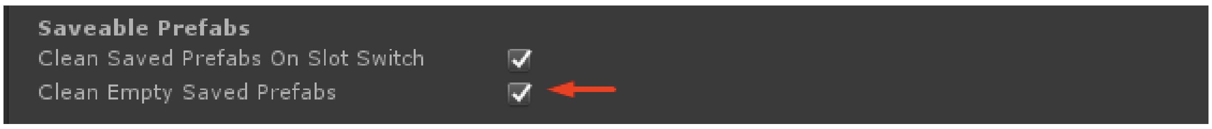
You can then choose to write this data to a specific slot by changing the slot by calling:

```
SaveMaster.SetSlot(slotIndex, false, keepActiveSaveData :true)
SaveMaster.WriteActiveSaveToDisk(true);
```

The **keepActiveSaveData** parameter makes it possible to overwrite a slot with your active save data. So you can overwrite existing slots with your current data. ***Do note, don't use keepActiveSaveData: true if you only want to load a game. As this will overwrite the slot with nothing.***

Saved prefabs and automatic cleanup

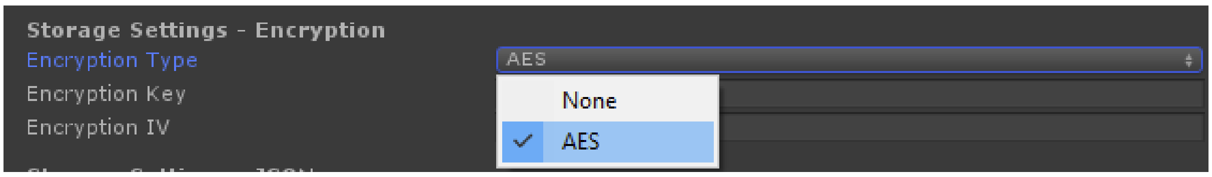
When you call **SaveMaster.SpawnSavedPrefab(InstanceSource.Resources, path)**
A new prefab gets spawned in the world that gets saved and respawned automatically when you load the scene again. In some cases a saved instance has nothing to save or load, by default these prefabs get cleaned up (Won't get saved, and will get cleaned up upon load). If you want to prevent this behaviour you have to disable **Clean Empty Saved Prefabs.**



Saveable Prefabs
Clean Saved Prefabs On Slot Switch ✓
Clean Empty Saved Prefabs ✓

Encryption

This is a new feature since release version 1.1. You are now able to encrypt your save file using AES encryption. However do keep in mind that this isn't a fully waterproof solution to keep the data secure. This is mainly a first line of defense, since users are able to decompile game code and resources with enough time. This just prevents casual users tampering with save files.

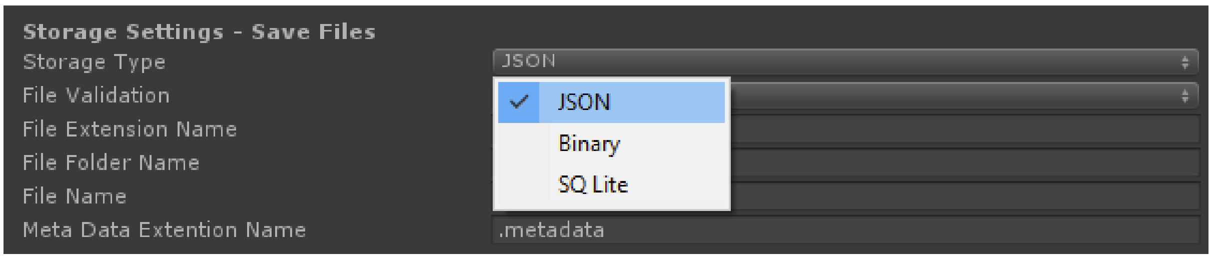


Storage Settings - Encryption
Encryption Type: AES
Encryption Key
Encryption IV

Once you set the encryption to AES. You can specify a Key and IV, these will be converted to a key under the hood. If you keep the fields blank, it will use a default Key and IV.

Do note, if you want to decrypt specific save games, you can do so by navigating to the Unity Item Menu. And going to **Window > Saving > Utility > Encryption**

Storage types



Storage Settings - Save Files
Storage Type: JSON
File Validation
File Extension Name
File Folder Name
File Name
Meta Data Extention Name: .metadata

By default the Save Component System uses JSON as a methodology for saving.
The reasoning for this is that JSON is easiest to read. Writing as Binary is faster then JSON, however it doesn't change too much. Since it still uses the same underlying data structures. Just the way it is written to a file has been changed. The SQLite option is still experimental. This option does not support all platforms, but it may be useful if you are working on a very big game.

Storing simple data, similar to PlayerPrefs

There are methods in place to store primitive data such as int, bool and string.
All you need to do is add the following.

```
using Lowscope.Saving;
```

After adding this you have access to the following methods to store your data.

```
SaveMaster.SetInt("myKey", 0);
SaveMaster.SetFloat("myKey", 0.0f);
SaveMaster.SetString("myKey", "text to save");
SaveMaster.SetBool("myKey", false);
```

And in order to retrieve the data you can use the following methods. The second parameter is not mandatory. You can use it to return a specific value in case your saved value is not found.

```
int myInt = SaveMaster.GetInt("myKey", defaultValue: 0);
float myFloat = SaveMaster.GetFloat("myKey", defaultValue: 0.0f);
string myString = SaveMaster.GetString("myKey", defaultValue: "");
bool myBool = SaveMaster.GetBool("myKey", defaultValue: false);
```

Examples of storing simple data

These are just some simple examples of things you could save using these methods. Personally I would still use the saveables system instead for things like achievements. However, if you ever have values you need between classes and scenes. Then this way of saving could be useful.

```
SaveMaster.SetString("gamemode", "hard");
SaveMaster.SetBool("achievement-100kills-completed", false);
SaveMaster.SetInt("achievement-100kills-progress", 35);
SaveMaster.SetFloat("game-volume", 0.55f);
```

Saving and loading your project manually

Depending on your game, you may want to have manual save events. To get stated you have to disable automatic saving. You can turn it off in the save settings. After you have configured the save settings. You can choose the following options, depending on the context of your project.

Saving & Writing to disk based on the active slot

This works only if you let players select a slot at the beginning of the game. Or if you load a default slot in the beginning. If you want the game to save using save locations or on a specific trigger, you can call this command to save all saveables in the scene and write the save data to disk:

```
SaveMaster.WriteActiveSaveToDisk(true);
```

The parameter indicates if we want to sync all saveables in the scene. If this is false then nothing in the active scene gets a save request. Which can be useful in specific scenarios.

Setting a slot and overwriting it

This is useful if you want to use a save select screen, where you can choose to overwrite a specific save slot. This command is enough to automatically change the slot, keep your save data and overwrite the active slot. **If you don't want to write anything yet, but want to change the slot you can add this parameter: writeToDiskAfterChange : false. It is preferable to have this option stay on. As the WriteToDisk event gets listened to by the SaveLastUsedScenes.**

```
SaveMaster.SetSlot(slot, false, keepActiveSaveData: true, writeToDiskAfterChange: true);
```

Removing a save slot and it's data.

In order to remove a save slot, which could be an option when selecting save slots.
The easiest way to remove a specific slot is to call the following command.

```
SaveMaster.DeleteSave(slotIndex);
```

You can also call this without parameters if you want to remove the active save.
If you do this, do not forget to change the slot to a valid slot. Since it will become invalid.

Loading and writing metadata

Since version 1.1 it is now possible to store Metadata for a save game. The reason for this introduction was to eliminate the need for loading entire save files to obtain basic save data. The Save Screenshot and Load Screenshot make use of this feature to store image data to show in a load sceen.

So in short, metadata allows you to write and load data to a separate file, which reduces load time and garbage collection. I will be using the **SaveScreenshotDisplayer** and **SaveScreenshot** code as an example on how to access the MetaData from a save file.

Writing to the metadata file

As you can see in the SaveScreenshotDisplayer code, I'm writing data to the following ids: **screenshot-width**, **screenshot-height** and **screenshot**

The first two being integers and one being a texture.

```
SaveMaster.SetMetaData("screenshot-width", resWidth.ToString());
SaveMaster.SetMetaData("screenshot-height", resHeight.ToString());
SaveMaster.SetMetaData("screenshot", screenShot);
```

The SetMetaData methods look like this. As you can see, string, texture2d and a byte array are available to write data towards. By modifying the slot index you can access data from a specific slot.

```
public static void SetMetaData(string id, string data, int slot = -1, string fileName = "")
public static void SetMetaData(string id, Texture2D data, int slot = -1, string fileName = "")
public static void SetMetaData(string id, byte[] data, int slot = -1, string fileName = "")
```

Loading from the metadata file

Loading from the metadata uses an out parameter. Meaning you have to specify it beforehand and reference it. The reason for this is because the method returns true or false, to indicate if it succeeded in retrieving the data.

```
string screenShotWidth = "";
if (SaveMaster.GetMetaData("screenshot-width", out screenShotWidth, slot))
{
        int.TryParse(screenShotWidth, out resWidth);
}

string screenShotHeight = "";
if (SaveMaster.GetMetaData("screenshot-height", out screenShotHeight, slot))
{
        int.TryParse(screenShotHeight, out resHeight);
}
```

Accessing savegame classes directly

In some rare cases you would want to access something in a specific save file. Or all save files. This is possible through the **SaveFileUtility** class. The SaveMaster class uses this class extensively for saving and loading, obtaining save names and obtaining available slots. Essentially, as the name suggests it focuses on managing save files.

The reason you are not able to get a reference to a savegame class directly is because it isn't meant to be directly manipulated by the users of this plugin. However, in cases that you do need to do this you can choose to use this class to obtain it.

Obtaining a save game based on slot

```
SaveFileUtility.LoadSave(slot, createIfEmpty : false);
```

Setting ID data

```
saveGame.Set(id, data, scene);
```

Getting ID data

```
saveGame.Get(id);
```

Removing ID data

```
saveGame.Remove(id);
```

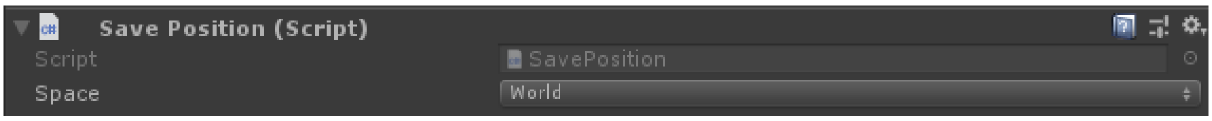
Wiping data from a specific scene

```
saveGame.WipeSceneData(sceneName);
```

Currently these are the only methods exposed outside of the classes. If you want to do things like enumerate over all the data, then I will advise you to modify the source directly. In the future there will be a tagging system instead of a scene system to obtain specific save information/remove specific information.

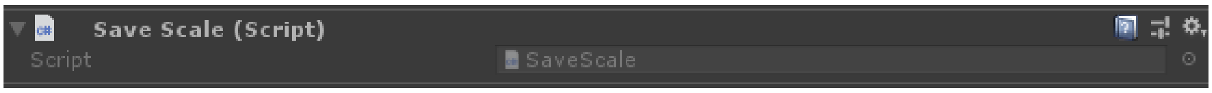
Component : Save Position

You can add this component to a game object that contains a **Saveable** component.
This component will save the active position, and it will set the position upon load during the **Awake()** step. You can configure the **Space** to either save the World or the Local position.


Save Position (Script)
Script SavePosition
Space World

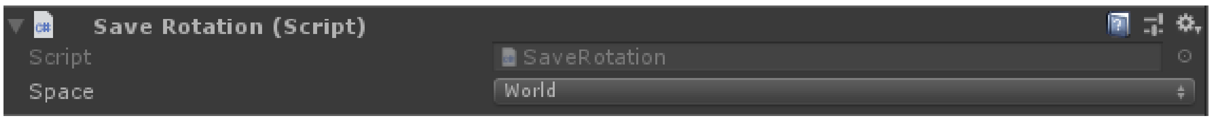
Component : Save Scale

You can add this component to a game object that contains a **Saveable** component.
This component will save the active local scale of the object, and it will set the scale upon load during the **Awake()** step.


Save Scale (Script)
Script SaveScale

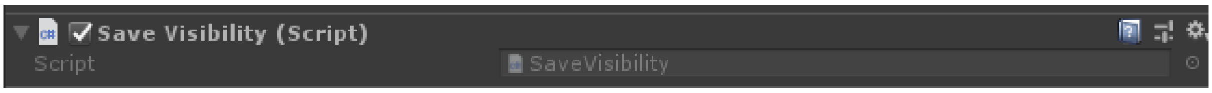
Component : Save Rotation

You can add this component to a game object that contains a **Saveable** component.
This component will save the rotation of the object, and it will set the rotation upon load during the **Awake()** step.


Save Rotation (Script)
Script SaveRotation
Space World

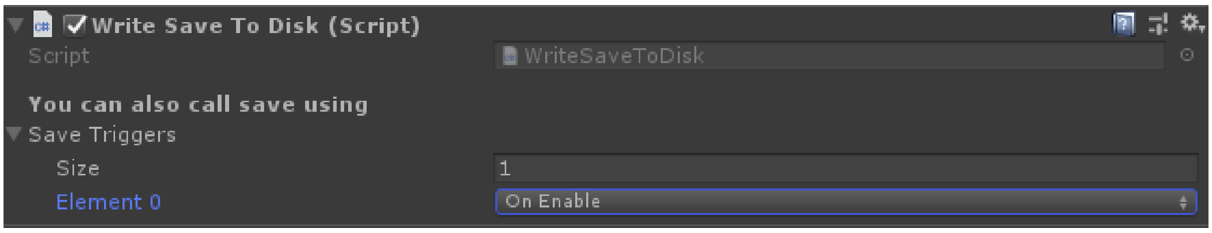
Component : Save Visibility

You can add this component to a game object that contains a **Saveable** component.
This component will save it's visibility. Based on the OnEnable and OnDisable methods.
So if the object gets destroyed or disabled during gameplay, it will be made invisible when loaded again. **The visibility will not be saved if it has been destroyed due to a scene change or if the game gets closed.**


Save Visibility (Script)
Script SaveVisibility

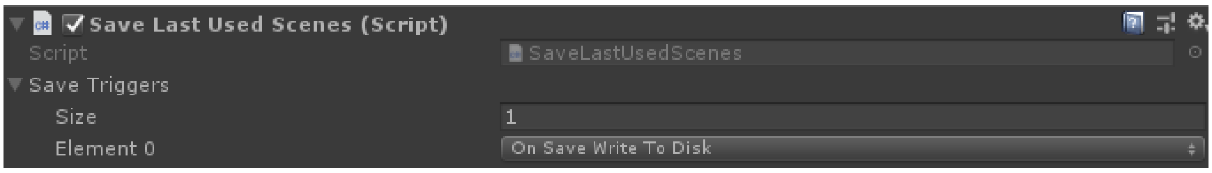
Component : Write To Disk

You can utilize this component if you want to easily trigger a write to disk event.
The intended use would be to have it in a scene where you want to trigger a save game. Or to have it disabled by default and enable it through an event. Or you can also manually trigger a function on it called **TriggerSave()**. When this component is triggered, it will automatically also sync all saveables in the scene.


Write Save To Disk (Script)
Script: WriteSaveToDisk
You can also call save using
Save Triggers
Size: 1
Element 0: On Enable

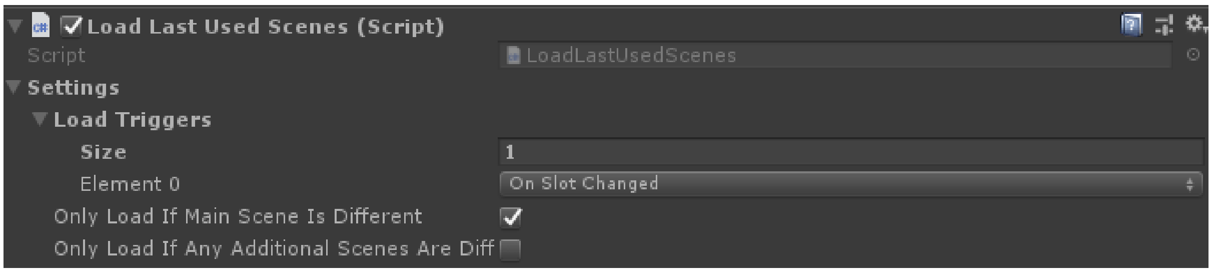
Component : Save Last Used Scenes

You can read more about this component on this page


Save Last Used Scenes (Script)
Script: SaveLastUsedScenes
Save Triggers
Size: 1
Element 0: On Save Write To Disk

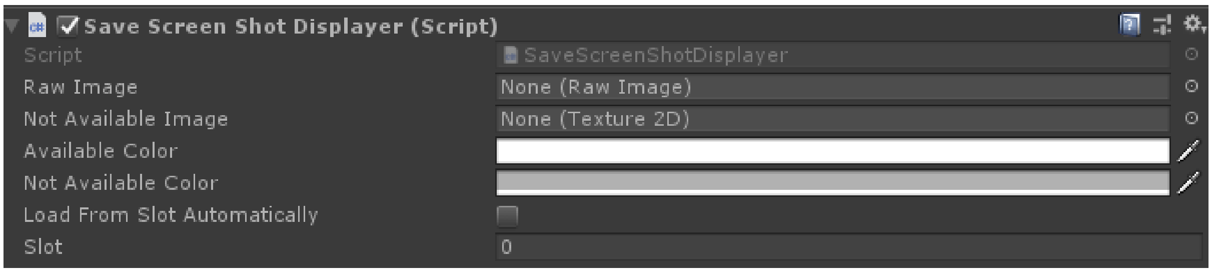
Component : Load Last Used Scenes

You can read more about this component on this page


Load Last Used Scenes (Script)
Script: LoadLastUsedScenes
Settings
Load Triggers
Size: 1
Element 0: On Slot Changed
Only Load If Main Scene Is Different: ✓
Only Load If Any Additional Scenes Are Diff: ☐

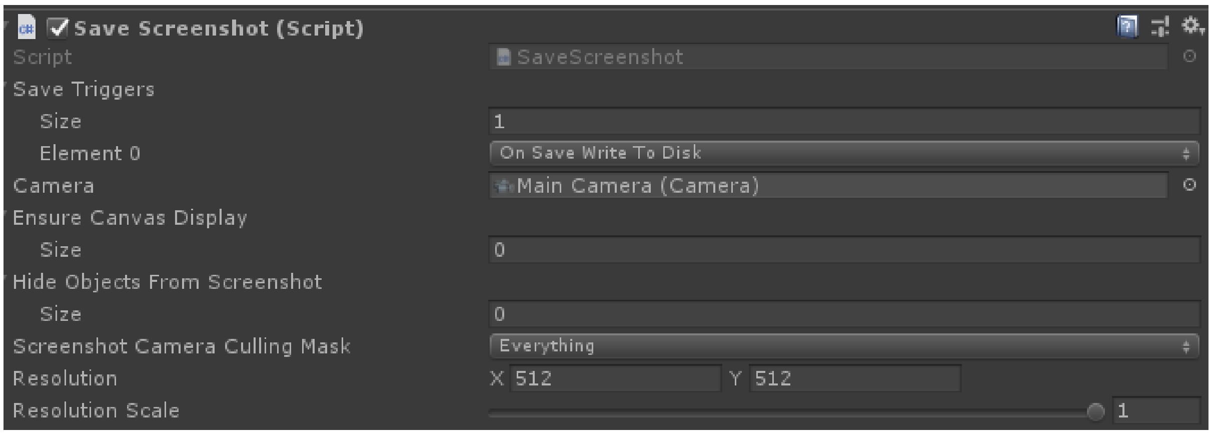
Component : Save Screenshot Displayer

You can read more about this component on this page


Save Screen Shot Displayer (Script)
Script: SaveScreenShotDisplayer
Raw Image: None (Raw Image)
Not Available Image: None (Texture 2D)
Available Color: ⬜
Not Available Color: ▨
Load From Slot Automatically: ☐
Slot: 0

Component : Save Screenshot

You can read more about this component on this page

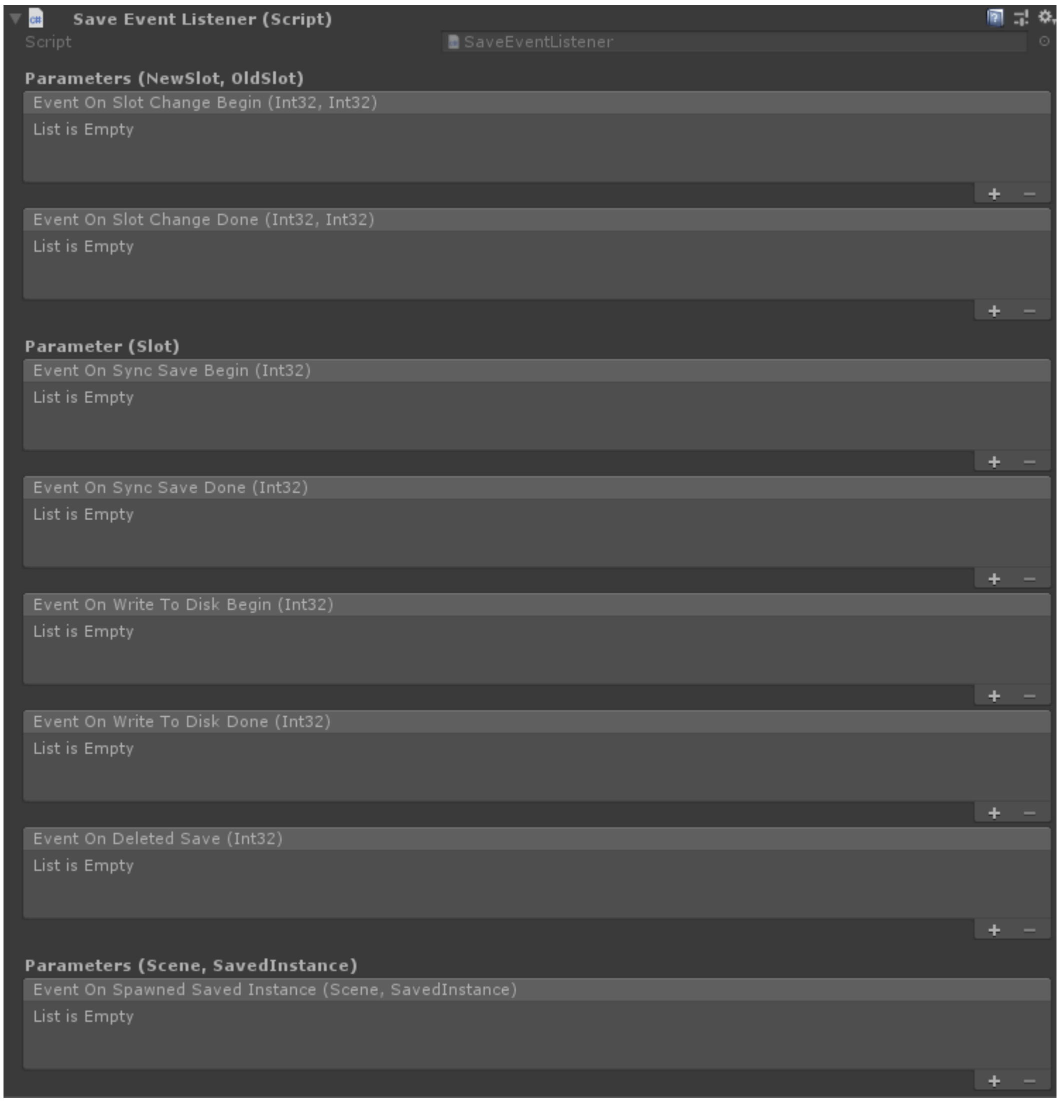


Save Screenshot (Script)

Script	SaveScreenshot
Save Triggers	
Size	1
Element 0	On Save Write To Disk
Camera	Main Camera (Camera)
Ensure Canvas Display	
Size	0
Hide Objects From Screenshot	
Size	0
Screenshot Camera Culling Mask	Everything
Resolution	X 512 Y 512
Resolution Scale	1

Component : Save Event Listener

This component allows you to listen to all common events emitted from the SaveMaster.



Save Event Listener (Script)

Script: SaveEventListener

Parameters (NewSlot, OldSlot)

Event On Slot Change Begin (Int32, Int32)
List is Empty

Event On Slot Change Done (Int32, Int32)
List is Empty

Parameter (Slot)

Event On Sync Save Begin (Int32)
List is Empty

Event On Sync Save Done (Int32)
List is Empty

Event On Write To Disk Begin (Int32)
List is Empty

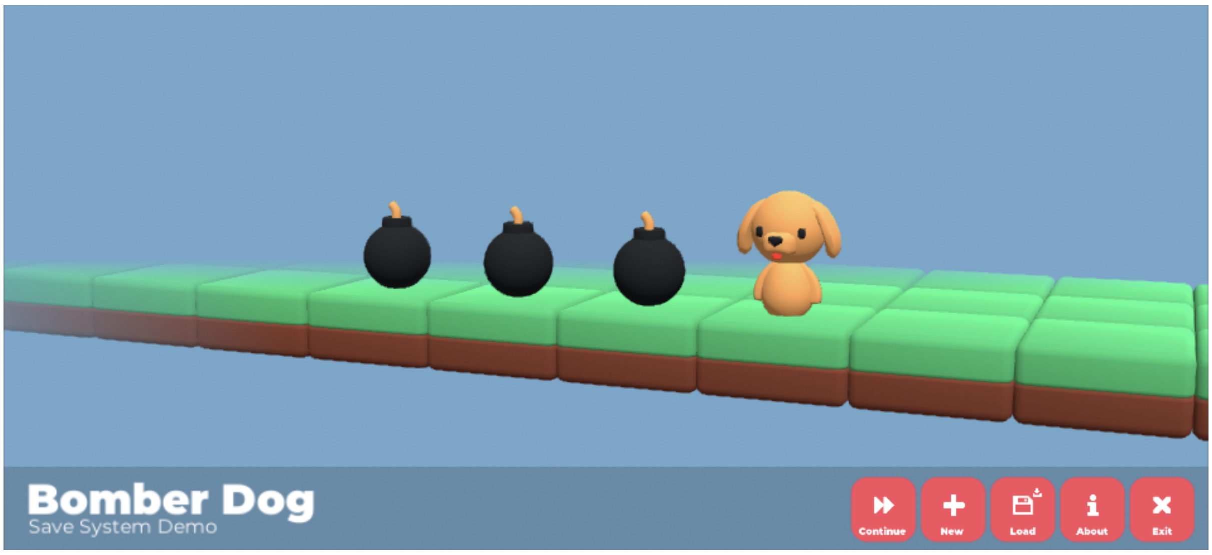
Event On Write To Disk Done (Int32)
List is Empty

Event On Deleted Save (Int32)
List is Empty

Parameters (Scene, SavedInstance)

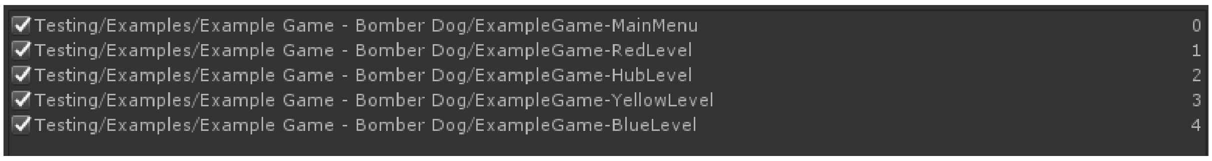
Event On Spawned Saved Instance (Scene, SavedInstance)
List is Empty

Bomber Dog - Sample game



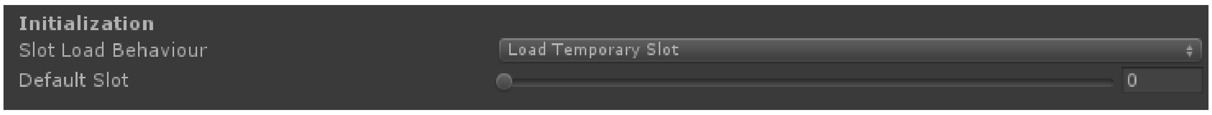
You can find the sample project for Bomber Dog in **Examples/Example Game - Bomber Dog.** If you want to test out the sample in the Unity Editor, there are a couple steps you need to take to get it fully functional. You can also try the game here.

1. **Add all the scenes from the game to your Build Settings**



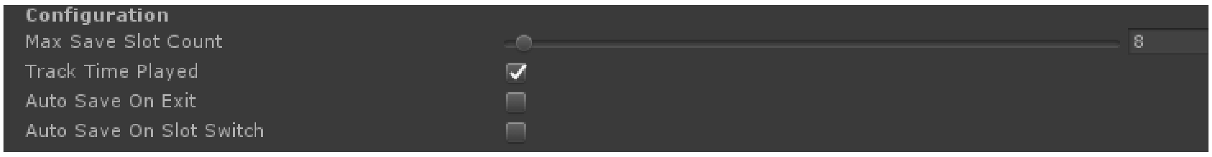
```
✓ Testing/Examples/Example Game - Bomber Dog/ExampleGame-MainMenu      0
✓ Testing/Examples/Example Game - Bomber Dog/ExampleGame-RedLevel      1
✓ Testing/Examples/Example Game - Bomber Dog/ExampleGame-HubLevel      2
✓ Testing/Examples/Example Game - Bomber Dog/ExampleGame-YellowLevel   3
✓ Testing/Examples/Example Game - Bomber Dog/ExampleGame-BlueLevel     4
```

2. **Set default slot loading to Load Temporary Slot**



```
Initialization
Slot Load Behaviour        Load Temporary Slot
Default Slot               0
```

3. **Turn off Auto Save On Exit and Auto Save On Slot Switch**

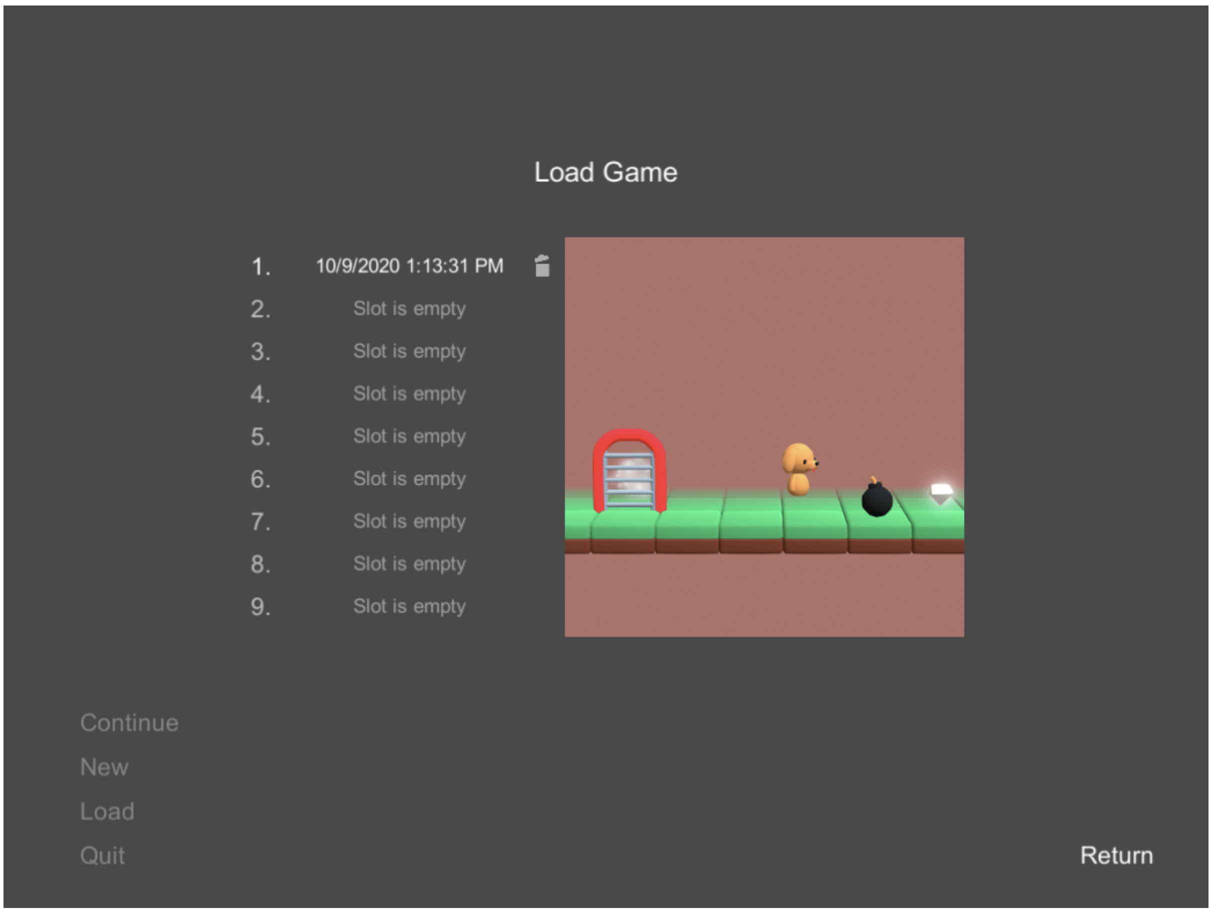


```
Configuration
Max Save Slot Count        8
Track Time Played          ✓
Auto Save On Exit          ☐
Auto Save On Slot Switch   ☐
```

After doing the steps above, just load the Main Menu scene and you are good to go. Loading individual scenes is also possible.

Sample - Main Menu

The component save system comes with a sample game menu that contains a load screen.



Scene & Prefab location

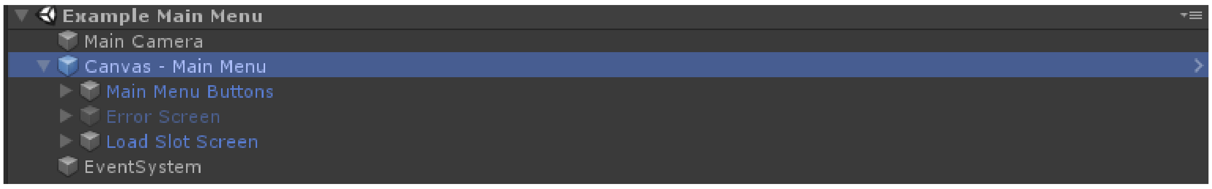
“Assets/Plugins/Lowscope/Component Save System/Examples/Main Menu and Pause Menu”
Within the Prefab folder in the scene location you can find the prefab to use it directly.
Scripts are also located within this folder.

Automatic scene loading on slot change

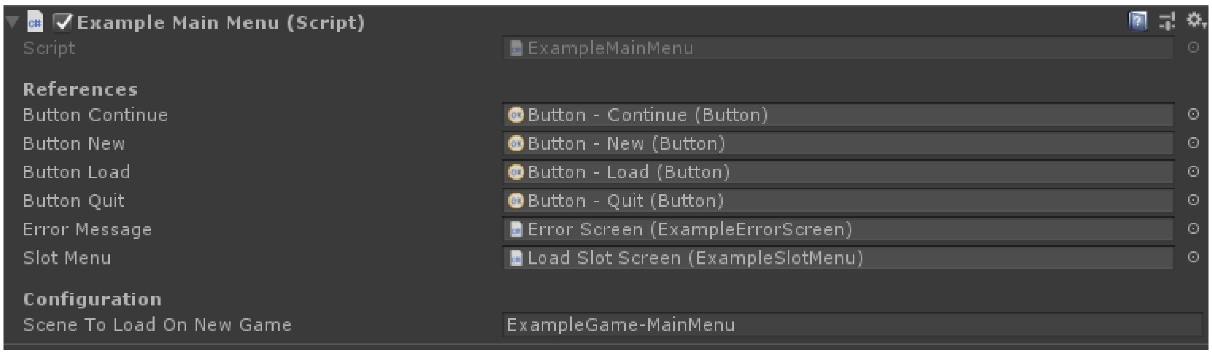
By default the **LoadLastUsedScenes** component is used for loading the last used scenes for a particular save file. This means you have to have the **SaveLastUsedScenes** component available in one of your main scenes. If you want to have a custom level loading solution, then I recommend removing the component and adding the **LoadSceneOnSaveTrigger** component instead that points to the scene that has custom loading. You can also look at the code of either component to see how it is done.

Configuring the “New Game” button

In order to change the scene you can load when you press the New Game button, navigate to the prefab or scene, and click on **Canvas - Main Menu**. Afterwards you can access the scene to load on the **Example Main Menu** component.



Example Main Menu
- Main Camera
- Canvas - Main Menu
  - Main Menu Buttons
  - Error Screen
  - Load Slot Screen
- EventSystem

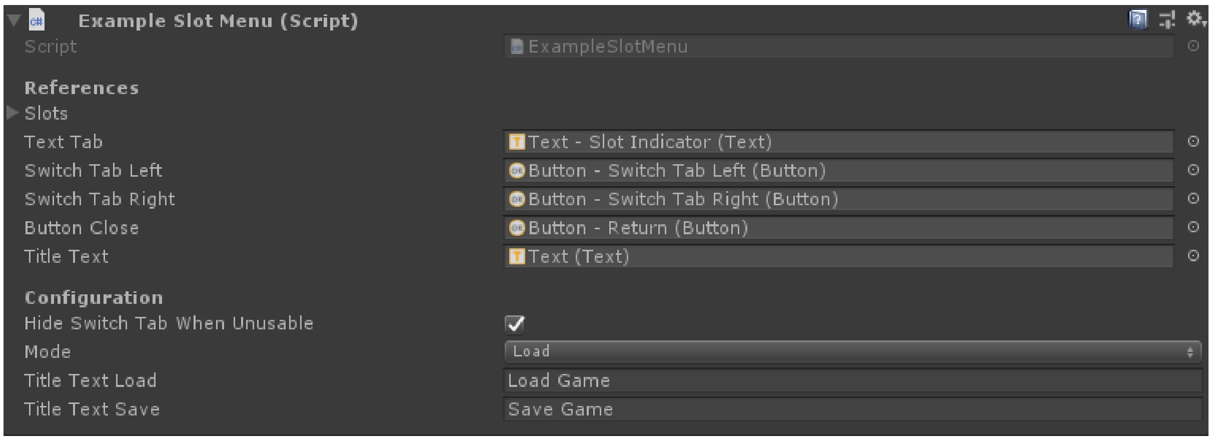


Example Main Menu (Script)

Script	ExampleMainMenu
References	
Button Continue	Button - Continue (Button)
Button New	Button - New (Button)
Button Load	Button - Load (Button)
Button Quit	Button - Quit (Button)
Error Message	Error Screen (ExampleErrorScreen)
Slot Menu	Load Slot Screen (ExampleSlotMenu)
Configuration	
Scene To Load On New Game	ExampleGame-MainMenu

Configuring the Slot Menu

For both the pause menu and main menu you can configure the Slot Menu.

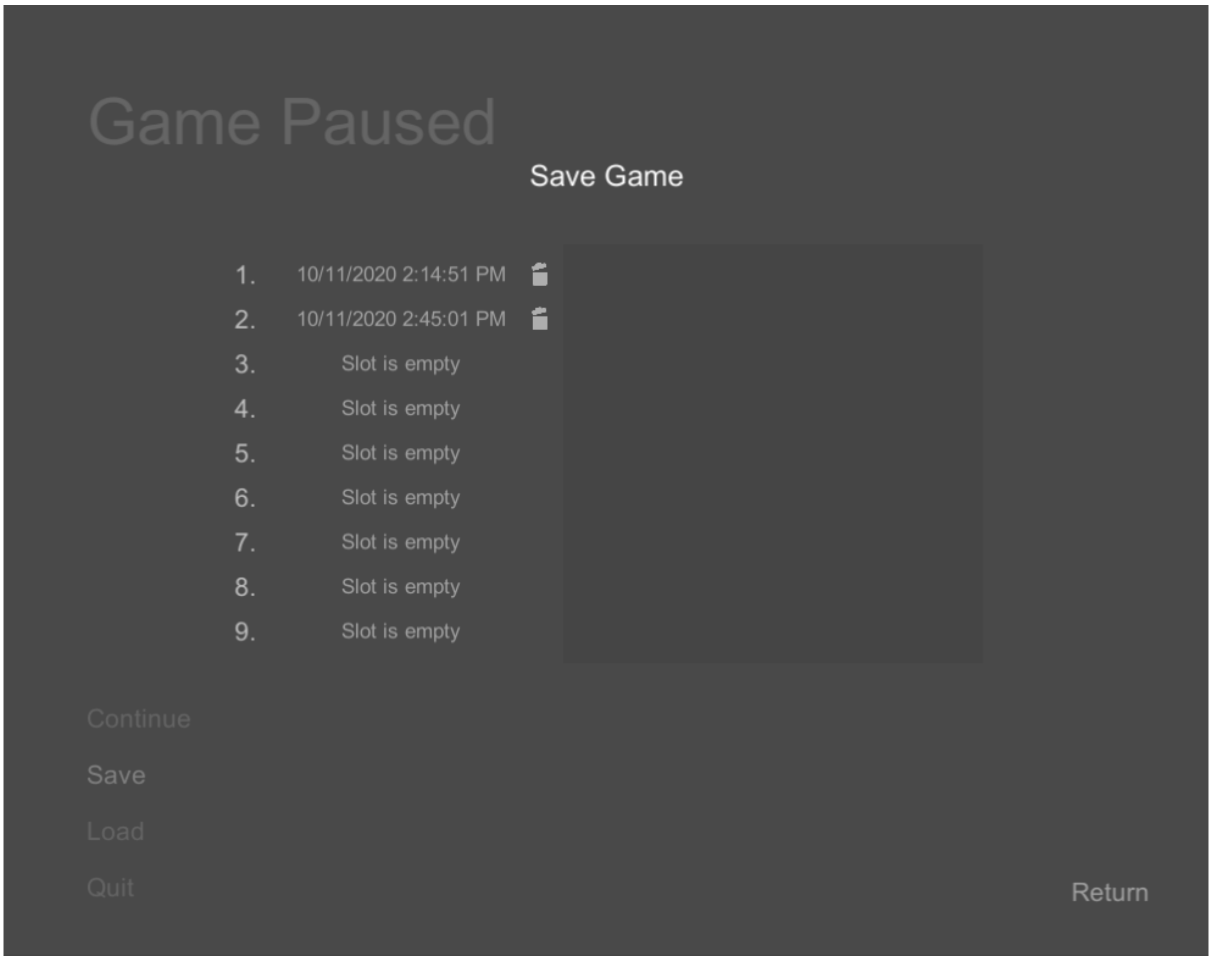


Example Slot Menu (Script)

Script	ExampleSlotMenu
References	
Slots	
Text Tab	Text - Slot Indicator (Text)
Switch Tab Left	Button - Switch Tab Left (Button)
Switch Tab Right	Button - Switch Tab Right (Button)
Button Close	Button - Return (Button)
Title Text	Text (Text)
Configuration	
Hide Switch Tab When Unusable	✓
Mode	Load
Title Text Load	Load Game
Title Text Save	Save Game

Sample - Pause Menu

The pause menu can be used as an example of how you could code your own pause menu. Or you could take the code and modify it yourself and use it as a base for a pause menu.



Automatic scene loading on slot change

By default the **LoadLastUsedScenes** component is used for loading the last used scenes for a particular save file. This means you have to have the **SaveLastUsedScenes** component available in one of your main scenes. If you want to have a custom level loading solution, then I recommend removing the component and adding the **LoadSceneOnSaveTrigger** component instead that points to the scene that has custom loading. You can also look at the code of either component to see how it is done.

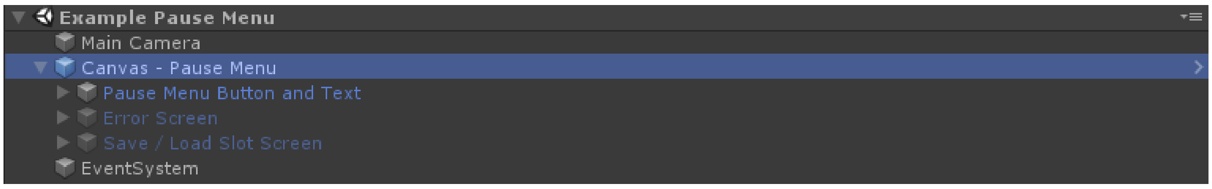
Configuring the Pause Menu

You can access the configuration by heading to the scene or prefab and selecting Canvas - Pause Menu. After selection, navigate to the Example Pause Menu script.

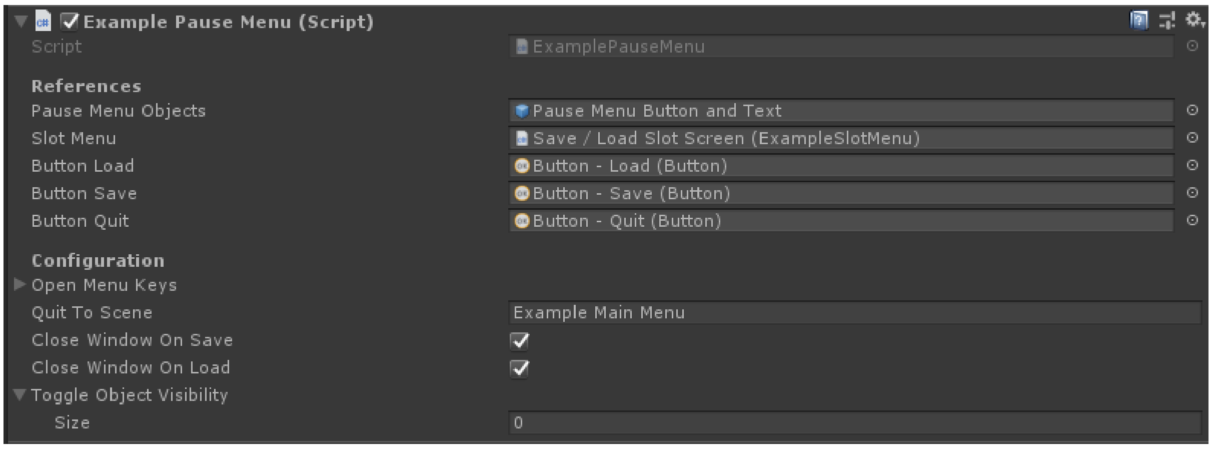
You can add hotkeys for opening the menu by unfolding the Open Menu Keys tab.
By default the Escape button is added to the list.

The **quit to scene** field can be changed accordingly so that it loads the main menu of your game. The **close window on save / load** can be used if you want the window to directly close after saving or loading.

The toggle object visibility option ensures objects have visibility switched when the window is open, and that the objects go back to its original state.



```
Example Pause Menu
    Main Camera
    Canvas - Pause Menu
        Pause Menu Button and Text
        Error Screen
        Save / Load Slot Screen
    EventSystem
```



Example Pause Menu (Script)

Script	ExamplePauseMenu
References	
Pause Menu Objects	Pause Menu Button and Text
Slot Menu	Save / Load Slot Screen (ExampleSlotMenu)
Button Load	Button - Load (Button)
Button Save	Button - Save (Button)
Button Quit	Button - Quit (Button)
Configuration	
Open Menu Keys	
Quit To Scene	Example Main Menu
Close Window On Save	✓
Close Window On Load	✓
Toggle Object Visibility	
Size	0

Contact

For questions, you can send an email to **info@low-scope.com**.

Licence