Save System Manual

Pixel Crushers Common Library

Contents

Chapter 1: Introduction to the Save System

The Pixel Crushers Common Library Save System provides a way to save and load games, and remember scene data across scene changes.

In This Manual

The rest of this manual contains these chapters:

- **Chapter 2: Save System** – *How to set up basic saving and loading*
- **Chapter 3: Scene Management** – *How to change scenes with persistence*
- **Chapter 4: Spawned Objects** – *How to keep track of objects created at runtime*
- **Chapter 5: Scripting** – *Scripting reference*

How to Get Help

We're here to help! If you get stuck or have any questions, please contact us any time at support@pixelcrushers.com or visit http://pixelcrushers.com.

We do our very best to reply to all emails within 24 hours. If you haven't received a reply within 24 hours, please check your spam folder.

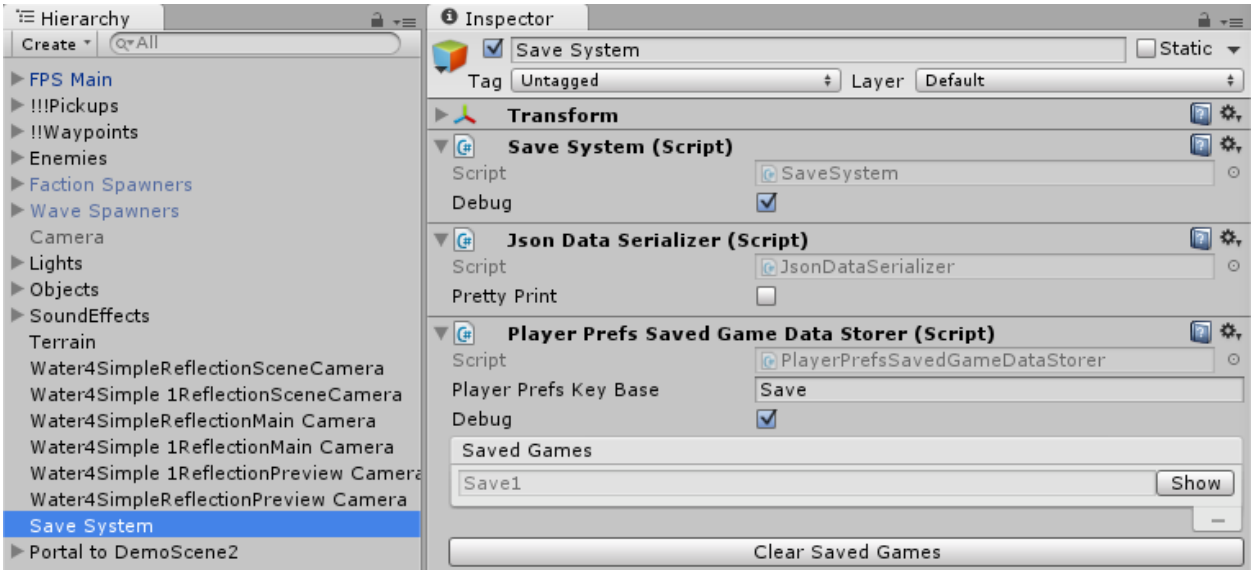
Chapter 2: Save System

This chapter describes how to set up the basic Save System.

Save System

The first step is to add a Save System component to the scene. This step is actually optional. The Save System will automatically create a Save System GameObject at runtime if one doesn't exist. However, you can manually add one if you want to customize it. To do so:

1. Create a GameObject, and add a **Save System** component. This GameObject will act as a *persistent singleton*, meaning it survives scene changes and typically there is only one instance.
2. Add a **Json Data Serializer** component. This component converts binary saved game data into a saveable format – in this case, JSON text. If you want to use a different serializer, you can add your own implementation of the DataSerializer class to the Save System GameObject.
3. Add a **Player Prefs Saved Game Data Storer** or **Disk Saved Game Storer** component. This component writes the serialized data to persistent storage such as PlayerPrefs or encrypted local disk files. Player Prefs Saved Game Data Storer is supported on all platforms. Disk Saved Game Storer is supported on standalone builds. If you want to store games a different way, you can add your own implementation of the SavedGameDataStorer class.



If you have a menu system, to load and save games you can assign the methods **SaveSystem.SaveGameToSlot** and **SaveSystem.LoadGameFromSlot** to your UI buttons' **OnClick()** events. Or in scripts you can use the equivalent static methods **SaveSystem.SaveToSlot** and **SaveSystem.LoadFromSlot** described in the next chapter.

Similarly, to change scenes without using the Scene Portal component, you can call **SaveSystem.LoadScene** in scripts or assign **SaveSystem.LoadScene** to your UI buttons' **OnClick()** events. Since the Save System GameObject might not be in all of your scenes, you can add a **Save System Methods** component and direct OnClick() to its LoadScene method instead.

Saver Components

Add these components to any GameObjects whose states you want to save:

Component	Function
Active Saver	Saves the active/inactive state of a GameObject. If the GameObject starts inactive, add this component to a different GameObject that's guaranteed to be active, and assign the target GameObject.
Multi Active Saver	Saves the active/inactive states of multiple GameObjects.
Animator Saver	Saves the state of a GameObject's animator.
Destructible Saver	Saves when a GameObject has been destroyed or disabled. The next time the game or scene is loaded, if the GameObject has previously been destroyed/disabled, this script will destroy/deactivate it again. It will also spawn a replacement destroyed version if a prefab is assigned.
Enabled Saver	Saves the enabled/disabled state of a component. If the GameObject starts inactive, add this component to a different GameObject that's guaranteed to be active, and assign the target component.
Multi Enabled Saver	Saves the enabled/disabled states of multiple components.
Position Saver	Saves a GameObject's position and rotation.

Every saver component needs a unique key under which to record its data in saved games. If the Key field is blank, it will use the GameObject name (e.g., "Orc"). If you tick **Append Saver Type To Key**, the key will also use the saver's type (e.g., "Orc_PositionSaver"). This is useful if the GameObject has several saver components.

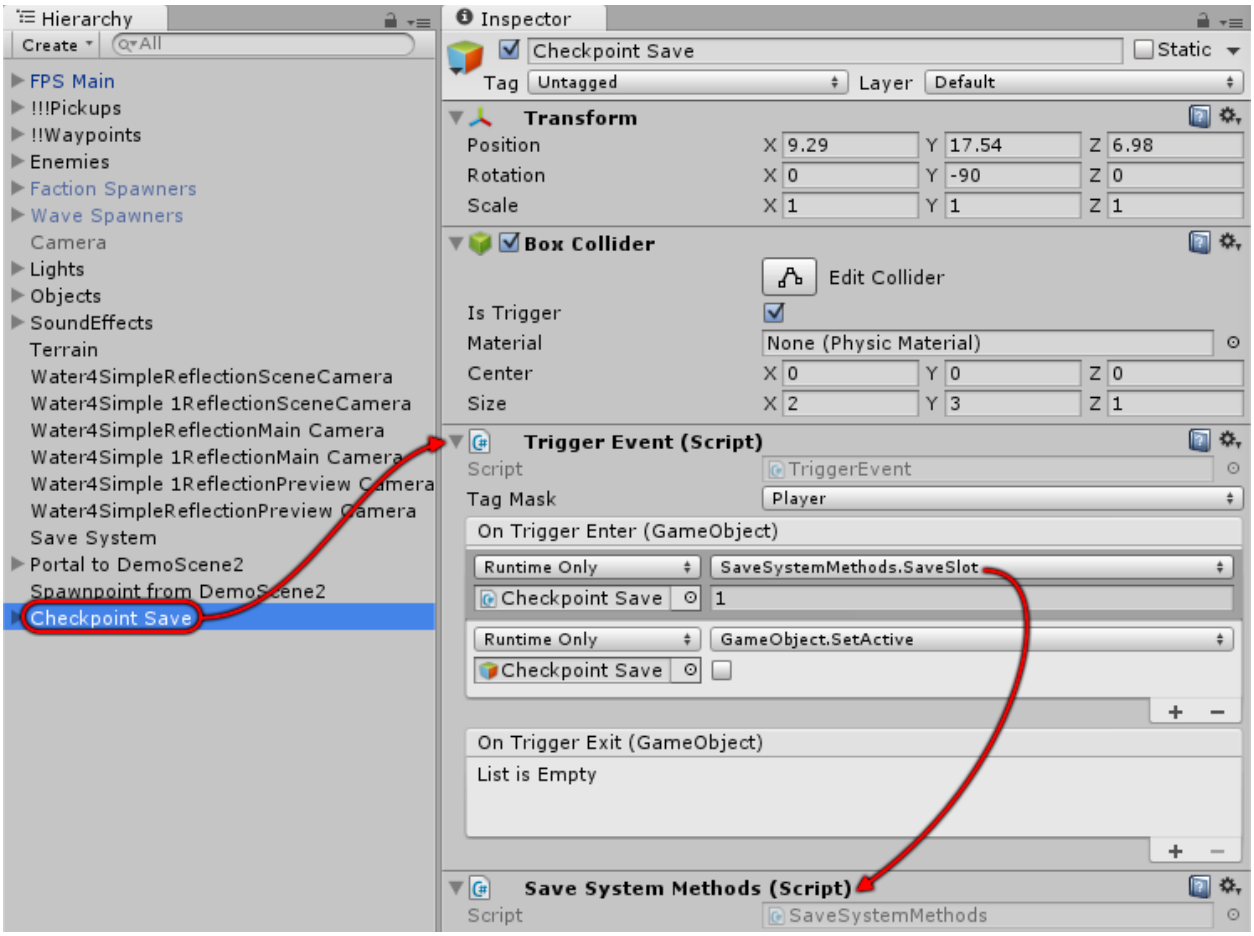
If GameObjects have the same name, you will need to assign unique keys. Otherwise they will all try to record their data under the same key, overwriting each other. To automatically assign unique keys to every saver in a scene, select menu item **Tools > Pixel Crushers > Common > Save System > Assign Unique Keys…**.

You can also write your own Saver components. The starter template script in Plugins / Pixel Crushers / Common / Templates contains comments that explain how to write your own Savers.

Checkpoint Saves

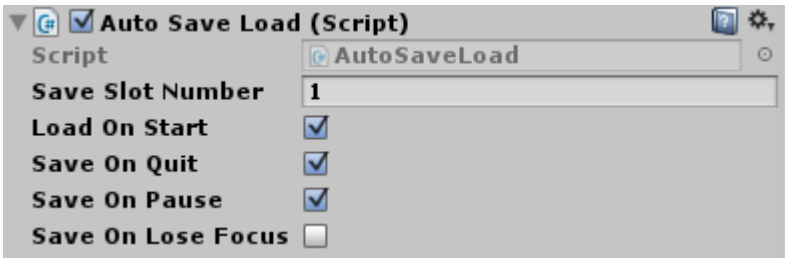
Checkpoint saves are trigger areas that automatically save the game when the player enters them. If you want to set up a checkpoint save:

1. Create a GameObject, and add a trigger collider.
2. Add a **Save System Methods** component.
3. Add a **Trigger Event** component.
   - Set the **Task Mask** to an appropriate mask, such as *Player*.
   - Configure the **On Trigger Enter** event to call SaveSystemMethods.SaveSlot with the saved game slot number you'd like to use.
   - You may also want to deactivate the GameObject in On Trigger Enter so it doesn't retrigger.



Auto Save Load

Mobile games typically auto-save when the player closes the game and auto-load when the player resumes the game. To add this behavior to your game, add an **Auto Save Load** component to the Save System:



Tick **Load On Start** to load the saved game (if it exists) on start, and **Save On Quit** to save the game when quitting the app.

Tick **Save On Pause** to also save the game when the player pauses/minimizes it. This way the game will be saved correctly if the player pauses the app and then kills it, instead of exiting it normally in the app itself.

Tick **Save On Lose Focus** to also save the game when the app loses focus.

Save System Events

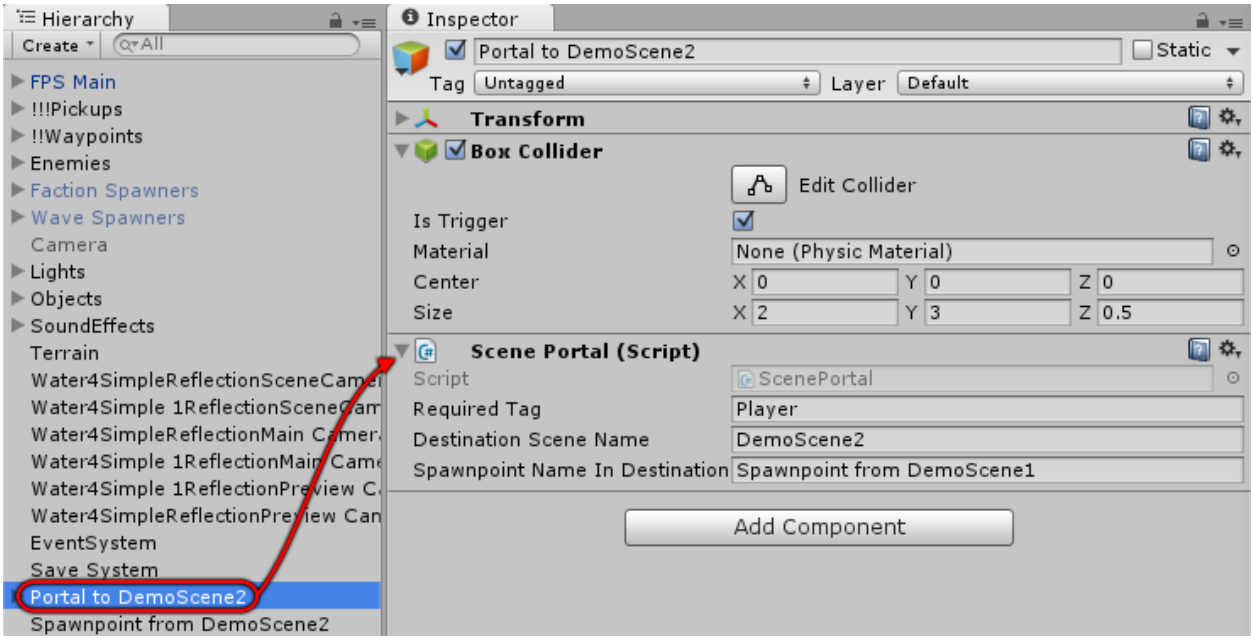
The Save System Events components allows you to hook up events in the inspector.

Chapter 3: Scene Management

Scene Portals

Scene Portals switch to another scene when the player enters them. To set up a Scene Portal:

1. Create a GameObject, and add a trigger collider.
2. Add a **Scene Portal component**.



Set the **Required Tag**, typically to *Player.*

Set **Destination Scene Name** to the scene that this portal leads to. Make sure the destination scene is in your project's build settings.

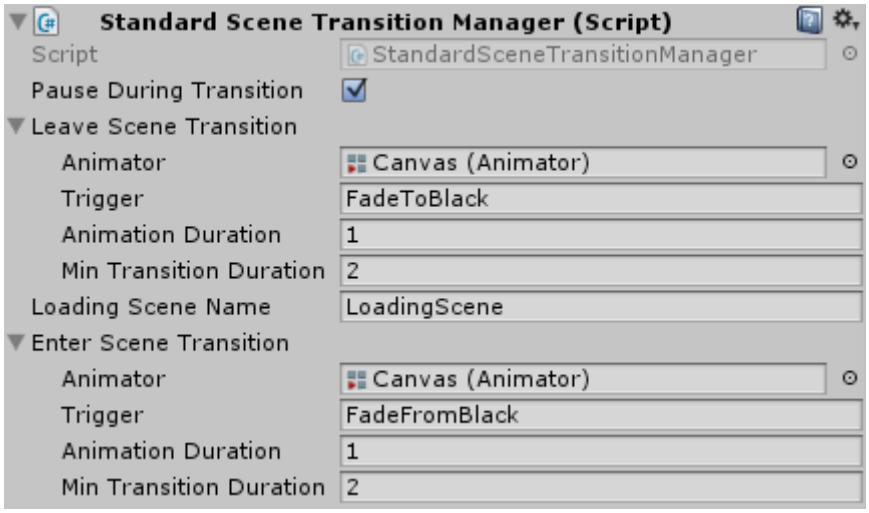
In the destination scene, create an empty GameObject where the player should appear. This is called a *spawnpoint*. In the original scene's Scene Portal component,set the **Spawnpoint Name In Destination Scene** field to the name of that spawnpoint.

Make sure your spawnpoint is far enough away from any scene portals in the destination scene so it won't immediately trigger another scene change.

If you don't want to use trigger colliders, you can manually call the **ScenePortal.UsePortal** method.

Scene Transition Manager

To play outro and intro animations, and/or show a loading scene while loading the next actual scene, add a **Standard Scene Transition Manager** to your Save System:



If **Pause During Transition** is ticked, make sure your Animator(s) are set to Unscaled Time.

If a Scene Transition Manager is present, the Save System will:

1. Set the Leave Scene Transition's animator trigger (if specified).
2. Load the loading scene (if specified).
3. Asynchronously load the next actual scene.
4. After the actual scene has been loaded, set the Enter Scene Transition's trigger (if specified).

Chapter 4: Spawned Objects

This section describes how to configure dropped objects to be saved in saved games.

Dropped objects are managed by a component called **Spawned Object Manager**. The Spawned Object Manager keeps track of objects that have been created in the scene at runtime. When you load a game, it re-instantiates the objects.

This is an overview of the configuration process:

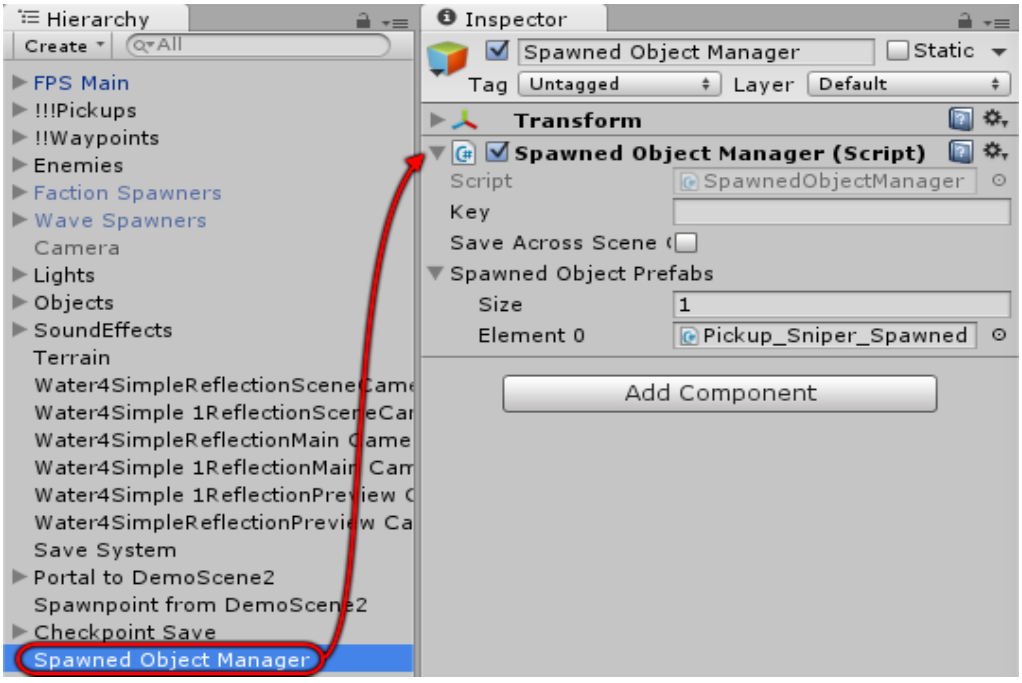
1. Add a **Spawned Object** component to your object prefabs.
2. Add a **Spawned Object Manager** to the scene, and assign the object prefabs to it.

Create Spawned Object Prefab

For this section, we use an example of tracking a sniper rifle. Find the prefab (e.g., Pickup_Sniper), and make a copy, for example named Pickup_Sniper_Spawned. Add a **Spawned Object** component to it. Repeat for all items that can be created in the scene.

Create Spawned Object Manager

Create an empty GameObject and add a **Spawned Object Manager** component as shown below. Each scene should have its own Spawned Object Manager with a unique key different from other scenes.



Add *all* Spawned Object prefabs that can be created in the scene to the **Spawned Object Prefabs** list. If a prefab is missing from the list, the Spawned Object Manager will not be able to re-spawn it when loading games or returning to the scene.

Chapter 5: Scripting

The Save System has a very simple scripting interface. The asset's online Scripting Reference contains complete API information if you want more details.

SaveSystem class

These common properties and methods are available though SaveSystem:

```
public static void LoadFromSlot(int slotNumber)
```
Loads a previously-saved game from a slot.

```
public static void SaveToSlot(int slotNumber)
```
Saves the current game to a slot.

```
public static void DeleteSavedGameInSlot(int slotNumber)
```
Deletes the data in a saved game slot.

```
public static void LoadScene(string sceneNameAndSpawnpoint)
```
Loads a scene, optionally positioning the player at a specified spawnpoint. The parameter is a string containing the name of the scene to load, optionally followed by "*@spawnpoint*" where "*spawnpoint*" is the name of a GameObject in that scene. The player will be spawned at that GameObject's position. This method implicitly calls RecordSavedGameData() before leaving the current scene and calls ApplySavedGameData() after loading the new scene.

```
public static SavedGameData RecordSavedGameData()
```
Returns a SavedGameData object containing the saved data from the current game.

```
public static void ApplySavedGameData([SavedGameData savedGameData])
```
Applies a SavedGameData object to the current game (in the current scene). If omitted, apply the most recently recorded data.

```
public static void LoadGame(SavedGameData savedGameData)
```
Loads a game previously saved in a SavedGameData object, including loading the saved scene.

```
public static void LoadScene(string sceneName, string spawnpointName = null)
```
Loads a scene, optionally positioning the player at a spawnpoint in the new scene.

```
public static void RestartGame(string startingSceneName)
```
Clears the current save game data and restart the game at the specified scene.

```
public static void ResetGameState()
```
Clears the current save game data without loading a new scene.

```
public static void LoadAdditiveScene(string sceneName)
```
Additively loads a scene into the current environment and tells its Savers to apply their states from the current saved game data.

```
public static void UnloadAdditiveScene(string sceneName)
```
Unloads a previously additively-loaded scene.

```
public int string version
```
Version number to include in save files. You can use it if your data changes between releases.

Saver class

Saver is the base class for any components that record data for saved games. You can create subclasses to extend the data that the Save System saves. To create a new saver subclass, copy Templates / SaverTemplate.cs to a new filename and edit it. The comments contain detailed instructions. You can also refer to other subclasses such as DestructibleSaver.cs and PositionSaver.cs for examples.

DataSerializer class

DataSerializer is the base class for data serializers that the Save System can use to serialize and deserialize saved game data. The default subclass is JsonDataSerializer.

SavedGameDataStorer class

SavedGameDataStorer is the base class for data storers. A data storer writes and reads a SavedGameData object to and from some storage location, such as PlayerPrefs or a disk file. The default subclass is PlayerPrefsSavedGameDataStorer, but the Save System also includes DiskSavedGameDataStorer which saves to encrypted disk files on supported platforms (e.g., desktop).

Events

You can register listeners for these C# events:

```
public static event SceneLoadedDelegate sceneLoaded = delegate { };
public static event System.Action saveStarted = delegate { };
public static event System.Action saveEnded = delegate { };
public static event System.Action loadStarted = delegate { };
public static event System.Action loadEnded = delegate { };
```

Save System Methods

To access Save System methods without scripting, such as in a UI Button's OnClick() event, add a **Save System Methods** component to the scene. This component exposes the methods of the SaveSystem class to the inspector.

Chapter 6: Serializers

The default serializer is JSON Data Serializer. However, you can remove this and add a Binary Data Serializer if you want to serialize to binary format. If your Savers use types that are not serializable, you will need to make a subclass of Binary Data Serializer and add serialization surrogates. For examples, see the scripts BinaryDataSerializer.cs, Vector3SerializationSurrogate.cs, and QuaternionSerializationSurrogate.cs.