Save System

for Opsive Character Controllers

User Guide

v1.0.x

Pixel Crushers

Save System for Opsive Character Controllers

Note: A Russian translation of a previous version of this manual was generously provided by a customer. You can access it here:
https://www.pixelcrushers.com/save-system-for-opsive-controllers/

Contents

Chapter 1: Welcome to the Save System....4
How to Get Help....4
Demo....4
Chapter 2: Configuration....5
Scene Setup....5
Player....5
Savers & Keys....6
Position, Active, Enabled, and Animator Savers....6
Pickups and Destructibles....6
Spawned Objects....8
Scene Portals....9
Checkpoint Saves....10
Save System....11
Addressables....11
Scene Transition Manager....12
Save System Events....13
Auto Save Load....14
Ultimate Inventory System Setup....15
Chapter 3: Scripting....16
Save System Methods....17
Chapter 4: Serializers....17
Chapter 5: Saved Game Data Storers....18

Chapter 1: Welcome to the Save System

Thanks for using the Save System for Opsive Character Controllers! This asset adds single player save support (save & load games and save changes across scene changes) to Opsive's character controllers. It requires one of these controllers:

- Ultimate Character Controller v3
- Third Person Controller v3
- UFPS: Ultimate First Person Shooter v3

It also includes integration with Ultimate Inventory System.

If you're using Ultimate Character Controller v2, contact us for a compatible package.

If you're using Opsive's older UFPS version 1, use the Save System for UFPS instead of this asset.

How to Get Help

We're here to help! If you get stuck, have questions, or want to request a feature, please contact us:

Email: support@pixelcrushers.com
Forum: https://pixelcrushers.com/phpbb
Discord: https://discord.gg/Nt9dVj

Note: This asset is developed and supported by Pixel Crushers, not Opsive.

Demo

To play the demo:

1. Import an Opsive character controller and the Save System for Opsive Character Controllers.
2. Add these scenes to your build settings:
    - Pixel Crushers ▸ Save System for Opsive ▸ Demo ▸ DemoScene
    - Pixel Crushers ▸ Save System for Opsive ▸ Demo ▸ DemoLoadingScreen
3. Play DemoScene. You can press Escape to open the demo menu containing buttons to save and load your game.

The rest of this manual describes how to set up your scenes and objects for saved games. It also covers other features that aren't included in the demo scene.

Chapter 2: Configuration

Follow the steps in this chapter to set up your project with the Save System for Opsive Character Controllers. You can apply the Save System to prefabs as well as scene GameObjects.

Scene Setup

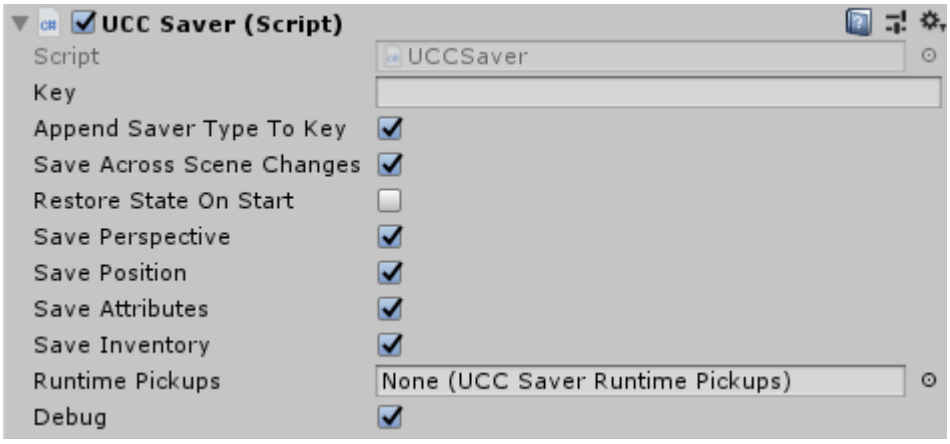
Add the **Save System** prefab from Pixel Crushers ▸ Save System for Opsive ▸ Prefabs to your starting scene. You can optionally add it to other gameplay scenes, too. But it must be in the first scene from which you will load a game (e.g., main menu scene).

If you want to add the demo menu to your own scene, add the **Save System Test Menu** prefab.

If you are also using Ultimate Inventory System, please see the additional setup steps in Ultimate Inventory System Setup.

Player

Add a **UCC Saver** component to your player.



This will save the player's stats, inventory, position, and current perspective if using a controller that can switch between first and third person perspective.

Data in saved games are stored under keys. You can specify a value in the **Key** field, or just leave it blank to default to the GameObject's name (e.g., Nolan). Make sure **Save Across Scene Changes** is ticked.

If the player can pick up items that aren't in the player's initial inventory list:

1. Right-click in the Project view and select Create > Pixel Crushers > Dialogue System > UCC Saver Runtime Pickups. This will create a UCC Saver Runtime Pickups asset.
2. Identify any items that are used by runtime pickups but are not in the player's initial inventory list. Add them to the UCC Saver Runtime Pickups asset.
3. Assign the the UCC Saver Runtime Pickups asset to the UCC Saver component.

To log debug information to the Console, tick **Debug**.

If you're only interested in saving the player's information, that's it. The rest of the manual explains how to configure other features such as scene-changing portals, savers for other types of objects, checkpoint saves, and the scripting API.

Savers & Keys

The Save System uses components called "savers" to save and restore information. The player's UCC Saver component is one example. The Save System includes several other types, too.

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.

Position, Active, Enabled, and Animator Savers

To save a GameObject's position, add a **Position Saver**.

To save a GameObject's animator state, add an **Animator Saver**.

To save a GameObject's active/inactive state add an **Active Saver**. To save a component's enabled state, add an **Enabled Saver**. In both cases, put the saver on a GameObject that's guaranteed to be active. To save the state of a pickup or destructible, use the instructions below instead.

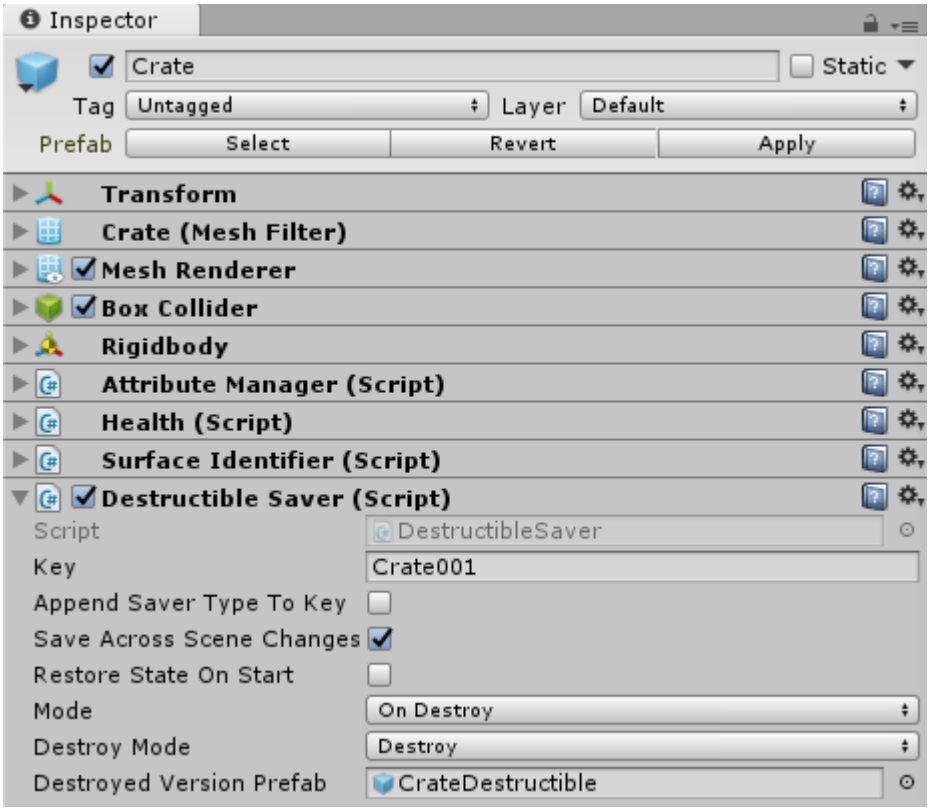
Pickups and Destructibles

To save the state of a pickup or destructible, add a **Destructible Saver**. Some objects, such as turrets, don't gracefully handle being destroyed; in this case, set **Destroy Mode** to *Deactivate* instead of *Destroy*.

If the destructible leaves behind a destroyed version, assign the destroyed version prefab to **Destroyed Version Prefab**. This typically matches the Health component's **Spawned Objects On Death** prefab. Alternatively, assign a spawned object version of the prefab to the Health component as described in the next section, *Spawned Objects*.

Tick **Save Across Scene Changes** to remember changes when leaving the scene and returning to it. Note that this will increase the size of the saved game file.

The screenshot on the next page shows the configuration of a destructible crate.


Inspector
Crate
Static
Tag
Untagged
Layer
Default
Prefab
Select
Revert
Apply
Transform
Crate (Mesh Filter)
Mesh Renderer
Box Collider
Rigidbody
Attribute Manager (Script)
Health (Script)
Surface Identifier (Script)
Destructible Saver (Script)
Script
DestructibleSaver
Key
Crate001
Append Saver Type To Key
Save Across Scene Changes
Restore State On Start
Mode
On Destroy
Destroy Mode
Destroy
Destroyed Version Prefab
CrateDestructible

Spawned Objects

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

Spawned objects are managed by a component called **Spawned Object Manager**. Although it has the word “Spawned” in the name, it has nothing to do with Opsive’s Respawner components. “Spawned” in this case means the game has instantiated an object into the scene, such as the clutter left behind after destroying a breakable crate. The Spawned Object Manager keeps track of these spawned objects. When you load a game, it re-instantiates the objects.

This is an overview of the configuration process:

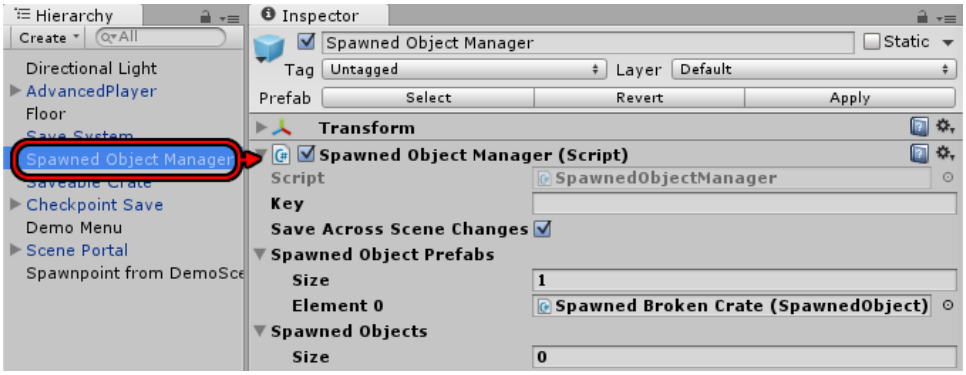
1. Create a copy of a drop/death spawn prefab, and add a **Spawned Object** component.
2. Add a **Spawned Object Manager** to the scene, and assign the prefab to it.
3. Configure objects to drop instances of this prefab instead of the regular Opsive prefab.

Create Spawned Object Prefab

Find the prefab (such as CrateDestructible), and make a copy, for example named Spawned Broken Crate. Add a **Spawned Object** component to it. Repeat for all items that the player can drop.

Create Spawned Object Manager

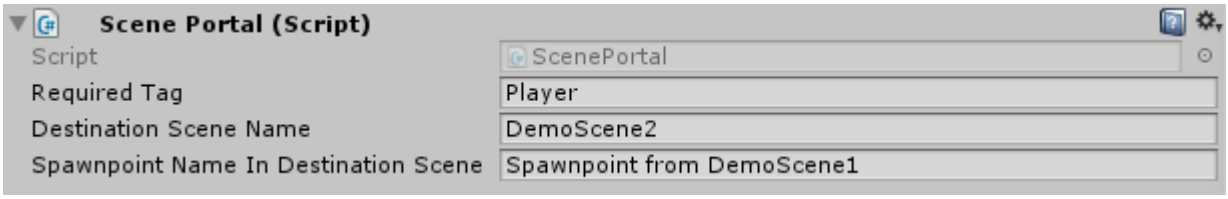
Drag the **Spawned Object Manager** prefab from the Prefabs folder into the scene, or create an empty GameObject and add a **Spawned Object Manager** component as shown below. Each scene should have its own Spawned Object Manager with a unique key unused in other scenes.



Add *all* Spawned Object prefabs that the player can drop to the **Spawned Object Prefabs** list. If a prefab is missing from the list, the Spawned Object Manager will not be able to respawn it when loading games or returning to the scene. You can also create and assign **Spawned Object List** assets.

Scene Portals

To set up a transition to another scene, drag the **Scene Portal** prefab from the Prefabs folder into your scene, or add a GameObject with a trigger collider and add a **Scene Portal** component.



If **Required Tag** is set, the portal will only respond to GameObjects that match the tag.
IMPORTANT: Opsive Character Controllers tag the root player GameObject as Player, but its collider is on a child GameObject Colliders > Capsule Collider. Set this child GameObject's tag 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.

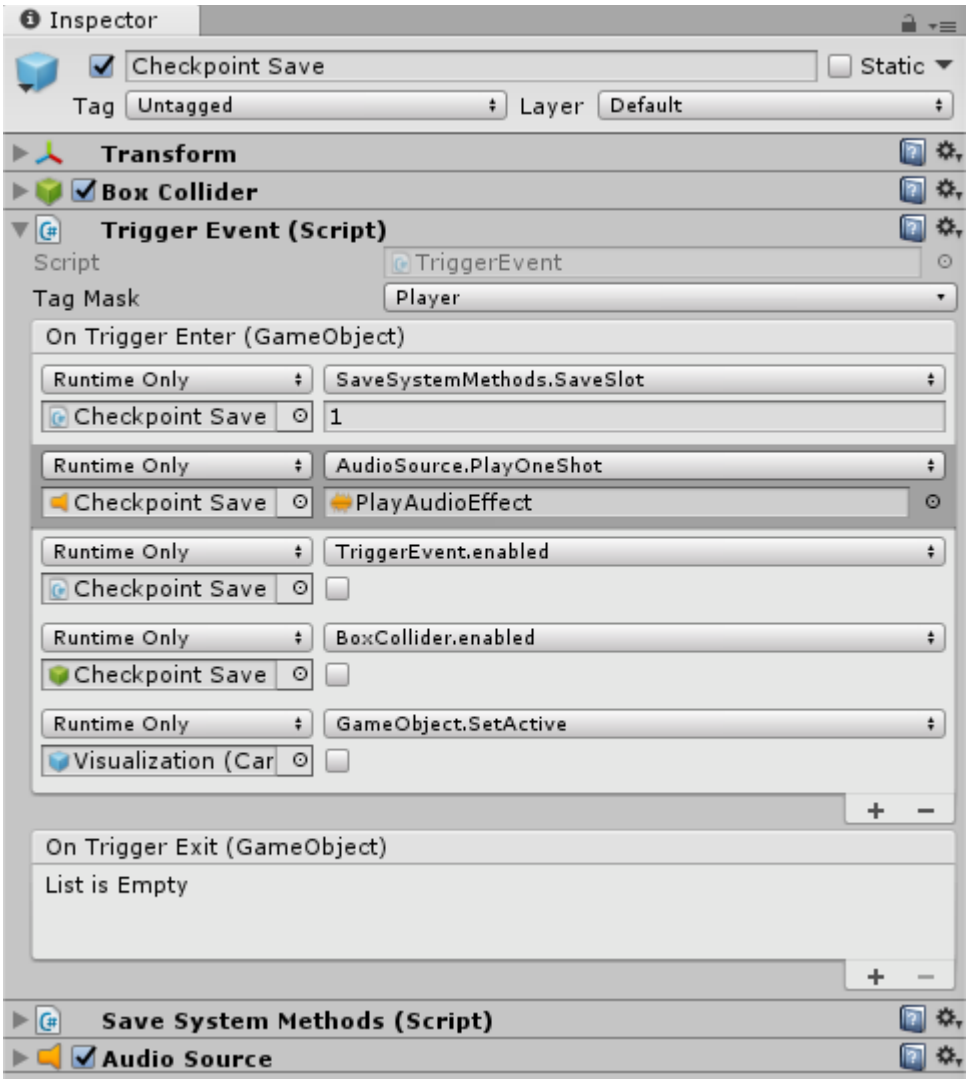
In the destination scene, create an empty GameObject where the player should appear. This is called a *spawnpoint*. In the original scene's Scene Portal component, set the **Spawnpoint Name In Destination Scene** field to the name of that spawnpoint.

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.

Checkpoint Saves

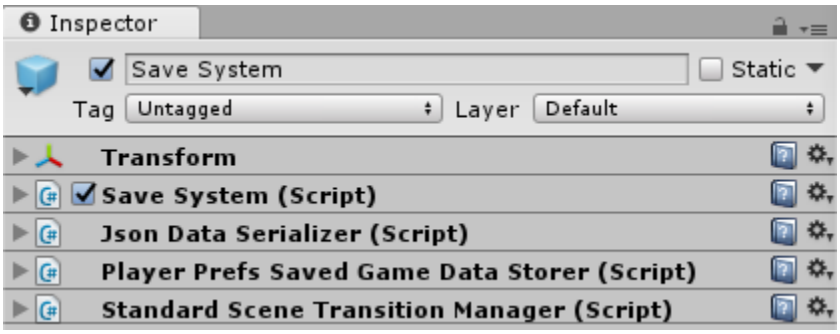
To set up a checkpoint save trigger, drag the **Checkpoint Save** prefab from the Prefabs folder into the scene, or add a GameObject with a trigger collider and add **Trigger Event** and **Save System Methods** components. The prefab is preconfigured to save in slot 1, play an audio clip, and hide the checkpoint.



If you add the components manually, configure the Trigger Event's **On Trigger Enter** event to call **SaveSystemMethods.SaveSlot**, and specify a slot number. Optionally, disable the trigger after the checkpoint save by adding events as shown above.

If you want the player to automatically respawn at the last checkpoint save when killed, replace the player's **CharacterRespawner** component with **CheckpointCharacterRespawner**. Set **Positioning Mode** to *None*; the save system will handle positioning according to the checkpoint save.

Save System



The Save System GameObject acts as a *persistent singleton*, meaning it survives scene changes and typically there is only one instance. If you change to a scene that also has a Save System GameObject, the original Save System GameObject will destroy the one in the new scene to ensure that there's only one.

The Save System relies on two types of components:

- **Data Serializer**: Converts binary saved game data into a saveable format.
- **Saved Game Data Storer**: Writes serialized data to persistent storage such as PlayerPrefs or local disk files.

By default, the Save System uses **Json Data Serializer**. If you want to use a different serializer, you can add your own implementation of the DataSerializer class to the Save System GameObject.

By default, the Save System uses **PlayerPrefs Saved Game Data Storer** to save games to PlayerPrefs. The Save System also ships with a **Disk Saved Game Data Storer** that saves games to encrypted local disk files; to use it, remove the PlayerPrefs Saved Game Data Storer (if present) and add Disk Saved Game Data Storer. If you want to store games a different way, you can add your own implementation of the SavedGameDataStorer class.

To load and save games you can call the methods described in the Scripting section to load and save games, and change scenes.

If you don't want to do any scripting, add a **Save System Methods** component to your UI buttons, and configure their OnClick() events to call the relevant methods such as LoadFromSlot or SaveToSlot.

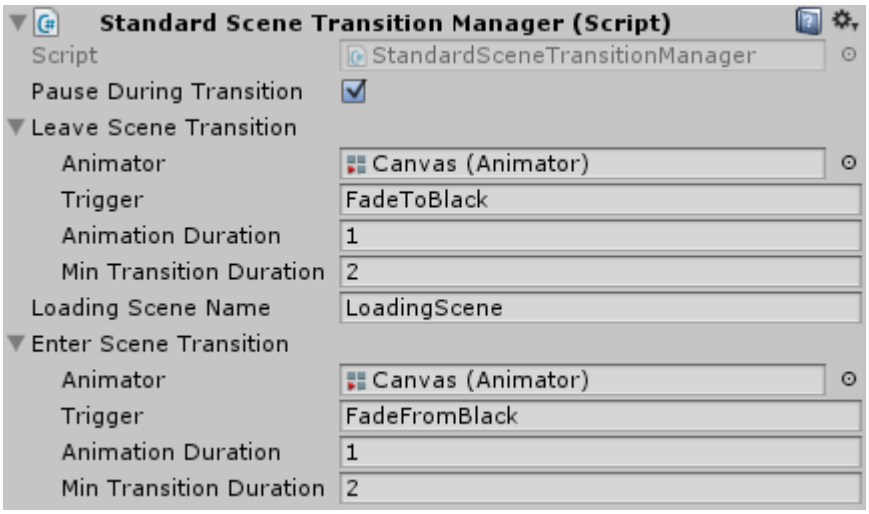
Addressables

If your scenes are Addressable assets, add the Scripting Define Symbol USE_ADDRESSABLES to tell the Save System to load scenes from Addressables if they're not in the build settings list.

The Save System also has an optional Scene Transition Manager, described below.

Scene Transition Manager

The Save System has a **Standard Scene Transition Manager** component, which lets you play outro and intro animations, and/or show a loading scene while loading the next actual scene. It's optional. If you don't want to use it, you can remove it.



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).

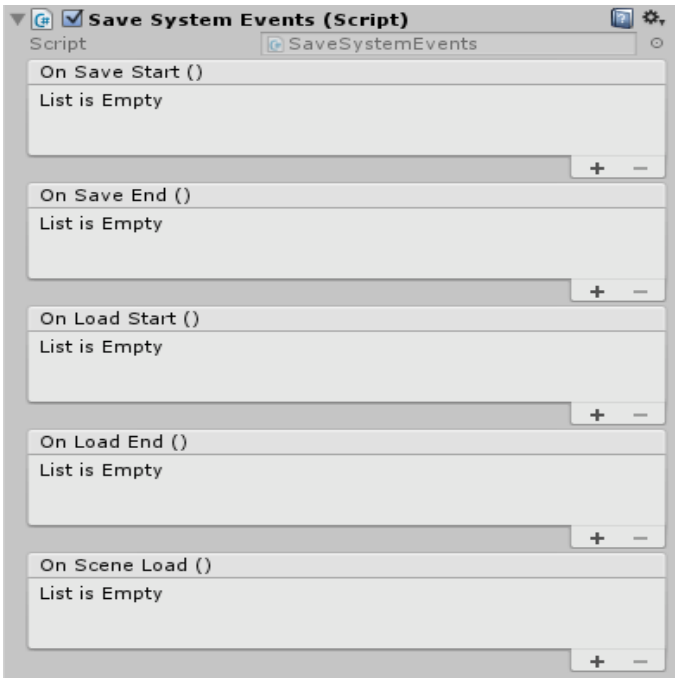
The default Save System prefab includes a one second fade-to-black when leaving the old scene and a one second fade-from-black when entering the new scene. This allows the Save System to reposition GameObjects in the new scene while it's covered in black.

If you want to use a different effect between scenes, you can write your own subclass of SceneTransitionManager and add it to your Save System instead.

Note: The Demo's Scene Transition Manager is configured to show an intermediate loading scene. If you simply want to fade to black, load the new scene, then fade in, simply unassign the loading scene name from the Scene Transition Manager.

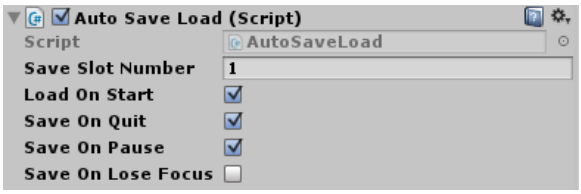
Save System Events

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



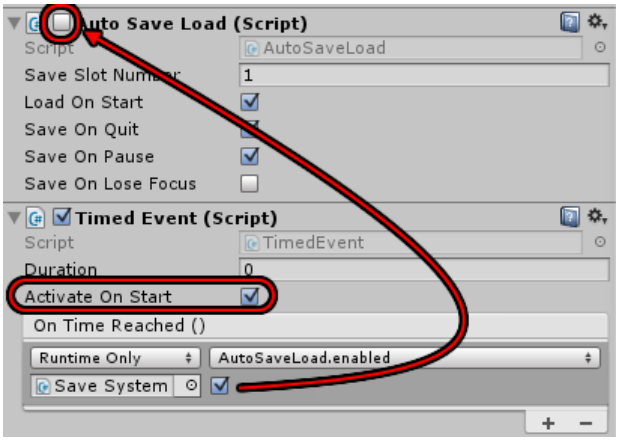
Auto Save Load

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.

If your game uses a splash screen, you may not want to enable Auto Save Load until the first scene actually loads. Otherwise, if the player minimizes the app during the splash screen, the Auto Save Load component could save the game in its initial state, before it has loaded its previously-saved data. To set this up, disable Auto Save Load, and use a **Timed Event** component to enable it on Start:



Ultimate Inventory System Setup

To also tie Ultimate Inventory System into the Pixel Crushers Save System:

- Import **Save System for UIS.unitypackage**, located in Pixel Crushers / Save System for Opsive / Extras.
- On the UIS Game GameObject, untick the **Inventory System Manager** component's Dont Destroy On Load checkbox. This will prevent duplicate copies when loading saved games.
- On the Save System GameObject, add a **UIS Saver** component.
- If you want to save the positions of items in a grid such as the player's inventory, add an **Inventory Grid Saver** component to the Inventory Grid in the UI.
- On the player, use an **Inventory Bridge Saver** component but do *not* add an Inventory Saver.
- On the player's **UCC Saver** component, untick Save Inventory since UIS will save inventory.
- Optional (*for testing only*): If you want to connect to UIS's inventory menu to the Pixel Crushers Save System's test menu script, add a **Redirect Save Load Button To Test Menu** component to the UIS MenuCanvas's SaveLoadButton GameObject. Then assign the UIS SaveLoadMenu GameObject to it. Note: This script is not intended for release builds. Use it only for quick testing.

Chapter 3: 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 are common properties & methods available in the SaveSystem class:

```
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.

```
public static void LoadGame(SavedGameData savedGameData)
```
Loads a game previously saved in a SavedGameData object.

```
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 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. A starter template with detailed comments is provided in Plugins ▶ Pixel Crushers ▶ Common ▶ Templates ▶ SaverTemplate.cs . 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 in the SaveSystem class:

```
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 4: 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.

If you want to serialize to a different format, you can write a new subclass of DataSerializer.

Chapter 5: Saved Game Data Storers

The default saved game data storer is PlayerPrefs Saved Game Data Storer. However, you can add a Disk Saved Game Data Storer if you want to save games to local disk files. Both of these save game data storers support encryption.

If you want to save to a different type of storage, such as a database or over a network, you can write a new subclass of SavedGameDataStorer.